Design and Performance of Interconnect-Efficient LDPC Codes with FPGA Implementation

BY

BADR HAMAD AD'DOHAN

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

TELECOMMUNICATIONS ENGINEERING

May 2008

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **BADR HAMAD AD'DOHAN**

Under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN **TELECOMMUNICATIONS ENGINEERING**

Thesis Committee

10/6/2009

Dr. Mohamad A. Landolsi

Thesis Advisor

Dr. Aiman H. El-Maleh

Thesis Co-advisor

13 JUN 2009

Dr. Samir H. Abdul-Jauwad

Department Chairman

Dr. Maan A. Kousa

Member

Dr. Abdallah S. Al-Ahmari

Member

Dr. Salam Adel Zummo

Dean of Graduate Studies

24/6/09

Date:

Dr. Ali H. Muqaibel

Member

To my parents

Acknowledgment

I would like to thank my advisors; Dr. Adnan Landolsi and Dr. Aiman El-Maleh for their invaluable help, patience and generous support throughout the course of this thesis. I am very grateful for their guidance and advising without which I wouldn't be able to finish my degree.

I would like also to thank my committee members; Dr. Maan Kousa, Dr. Abdallah Al-Ahmari and Dr. Ali Muqaibel for their valuable comments and support.

I would like to thank King Fahd University of Petroleum and Minerals for supporting this research.

Thanks are due to the former EE department chairmen, Dr. Jamil Bakhashwain and Dr. Ibrahim Habiballah for their help and support. Thanks are also extended to the current EE department chairman Dr. Samir Abdul-Jauwad.

I thank all my friends, colleagues and relatives for their help and encouragement. Special thanks go to Dr. Esa Al-Ghonaim for his help and assistance and for allowing me to use his software. Special thanks go also to Mr. Hesham Al-Salman for his continuous encouragement.

I would like also to thank Dr. Saad Aiban and Dr. Mustafa Achoui for their very appreciated advices that they gave me during some difficult times.

Finally, the most important thanks go to my father, mother and brothers for their generous love, support and encouragement that they have given me throughout the course of my academic career.

Table of Contents

Acknowledgment iii

List of Tables vii

List of Figures viii

Abstract x

خلاصة xi

Chapter 1 Introduction asdf 1

1.1LDPC in brief 1

1.2 Thesis Motivation 4

1.3 Thesis Contributions 4

1.4 Thesis Organization 5

Chapter 2 Encoding and Decoding of LDPC Codes 7

2.1 Representation of LDPC Codes 8

2.1.1 Matrix Representation 8

2.1.2 Graphical Representation 9

2.2 Encoding of LDPC codes 12

2.3 Iterative Decoding Algorithm 13

2.3. Overview 13

2.3.2 Probability-Domain SPA Decoder 17

2.3.3 Log-Domain SPA Decoder 22

2.3.4 LDPC Decoder Complexity 26

Chapter 3 Finite-Precision Performance of Semi-Random LDPC Codes 28

3.1 Introduction 28

3.2 Structure of Semi-Random LDPC Codes 29

3.3 Fixed-Point Implementation of Semi-Random LDPC Codes 31

3.4 Simulation Results 31
3.4.1 Effect of Varying Dynamic Range M 34
3.4.2 Effect of Varying number of iterations It_{max} 37
3.4.3 Effect of Varying number of bits 37
3.4.4 Joint Optimization of Dynamic Range M and Resolution Bits *nbit* 38
3.5 Conclusion 41
Chapter 4 LDPC Codes: Design and Performance 42
4.1 Introduction & Motivation 42
4.2 Code Structure 42
4.3 Assessment of the Complexity Reduction Advantage of the Proposed Code 47
4.3.1 Qualitative Assessment 47
4.3.2 Quantitative Assessment 49
4.4 Performance over AWGN 53
4.5 Conclusion 54
Chapter 5 Design and Implementation of the LDPC Decoder 55
5.1 Introduction 55
5.2 Decoder Hardware Architecture 56
5.2.1 The Check Node Unit (CNU) 56
5.2.2 The Variable Node Unit (VNU) 58
5.2.3 The Combined Decoder 59
5.3 VHDL Modeling and Verification 61
5.4 FPGA Synthesis of different LDPC codes 63
5.4.1 Synthesis Results 63
5.4.2 Effect of the LDPC Code Structure on the Synthesis Results 71
5.4.3 Effect of the LDPC Code Block length on the Synthesis Results: 72

5.4.4 Throughput....73
5.5 Conclusion....74
Chapter 6 Summary of Results and Future Work....76
Appendix A VHDL Code for The LDPC Decoder....79
A.1 The Overall VHDL Model....79
A.2 The Main File....80
A.3 The Matrix File....85
A.4 The CNU Files....85
A.4.1 CNU3....85
A.4.2 CNU4....87
A.5 The VNU Files....90
A.5.1 VNU2....90
A.5.2 VNU3....91
A.7 The Number Format Conversion Files....93
Appendix B MATLAB Code for Generating H matrix of the proposed code....95
References....98

List of Tables

Table 3.1 Data of the 3D plots 40

Table 5.1 Xilinx VirtexE Synthesis Results for Different LDPC Codes 64

Table 5.2 Synthesis Results Comparison for n=64 71

Table 5.3 Synthesis Results of the Proposed code for different sizes 72

Table 5.4 Throughput of the 128-bit decoder for different parameters 74

List of Figures

Figure 2.1 a 5x10 **H** matrix....10

Figure 2.2 Tanner graph for the example code. 11

Figure 2.3 Message passing from variable node c0 to check node f2. 15

Figure 2.4 Message passing from check node f0 to variable node c4. 16

Figure 2.5 Message Passing Half-Iteration For The Computation of qij(b). 19

Figure 2.6 Message passing half-iteration for the computation of $r_{ji}(b)$. 20

Figure 2.7 Plot of the ∅(x) function. 25

Figure 3.1 Illustration of The Quantization procedure. 33

Figure 3.2 Performance for different dynamic ranges *M*....34

Figure 3.3 Performance for different values of *M* at SNR=3.5 35

Figure 3.4 Histogram of LLR messages. 36

Figure 3.5 Performance for different maximum iterations It_{max}. 37

Figure 3.6 Performance for different bit precisions *nbit*....38

Figure 3.7 3D view of the performance as a function of *nbit* and *M*....39

Figure 3.8 Another 3D view of the performance as a function of *nbit* and *M*....39

Figure 4.1 Block diagram of the general structure of the code....43

Figure 4.2 Cell with size 16x8 44

Figure 4.3 Cell with size 32x16 44

Figure 4.4 A 3x3 Lattice composed by 32x16 cells....46

Figure 4.5 Interconnection complexity of a (512,1024) random LDPC code. 48

Figure 4.6 Interconnection complexity of a (512,1024) proposed LDPC code. 48

Figure 4.7 Example of a (32,16) LDPC code layout. 49

Figure 4.8 Average Manhattan distance for different LDPC codes....52

Figure 4.9 Maximum Manhattan distance for different LDPC codes. 52

Figure 4.10 Performance of the proposed code compared to two similar codes. 53

Figure 5.1 Check Node Unit (CNU) with 3 inputs. 57

Figure 5.2 the Variable Node Unit (VNU) with 3 inputs. 58

Figure 5.3 the Combined Decoder Architecture. 60

Figure 5.4 Structure of The VHDL model. 62

Figure 5.5 Overall view of synthesized proposed LDPC code with N=64.....65

Figure 5.6 Closer view of the synthesized proposed LDPC code with N=64 66

Figure 5.7 Overall view of the synthesized proposed LDPC code with N=128. 67

Figure 5.8 Closer view of the synthesized proposed LDPC code with N=128 68

Figure 5.9 Overall view of the synthesized random LDPC code with N=64. 69

Figure 5.10 Closer view of the synthesized random LDPC code with N=64. 70

Abstract

Name: BADR HAMAD AD'DOHAN

Title: Design and Performance of Interconnect-Efficient LDPC Codes with FPGA Implementation

Major Field: Telecommunications Engineering

Date: May 31st, 2008.

LDPC codes are a class of linear block codes that have gained a lot of attention in the last decade and have shown near-capacity performance on different channels. This thesis discusses several issues concerning efficient implementation of LDPC decoders. Fixed-point simulation is performed on Semi-Random LDPC codes in order to determine the effect of changing some hardware parameters on the performance. This is done to help in selecting a good compromise before getting into the actual implementation. A new structured LDPC code is proposed with the target of reducing hardware complexity. The proposed code is lattice-structured in a way that reduces the maximum wire length and is scalable. The LDPC decoder has been modeled in VHDL. This model is generic and can be used for any LDPC code. This model has been used to demonstrate the hardware advantages of the proposed LDPC code in comparison with a random code.

خلاصة

الإسم: بدر بن حمد بن إبراهيم الدوهان

عنوان الرسالة: تصميم و معايرة أداء رموز LDPC ذات الكفاءة التشبيكية مع تنفيذها بإستخدام FPGA .

الحقل التخصصي: هندسة الإتصالات

التاريخ: 31 مايو 2008, 26 جمادى الأولى 1429هـ.

الرموز ذات مصفوفة فحص التماثل منخفضة الكثافة (LDPC)(فتمك) هي فئة من رموز المجاميع الخطية حظيت بإهتمام كبير في العقد الأخير و أظهرت أداء مقارباً للسَّعة لكثير من قنوات الإتصال. هذه الرسالة العلمية تناقش عدة قضايا متعلقة بالتنفيذ الكفء لفك رموز (فتمك). سيتم إجراء محاكاة النقطة الثابتة على رموز فتمك شبه العشوائية من أجل تحديد أثر تغيير بعض مؤشرات الأجهزة على الأداء.هذا العمل سيساعد على اختيار حلول توفيقية جيدة لمؤشرات الأجهزة قبل المضي في التنفيذ الفعلي. تم اقتراح رموز فتمك هيكلية جديدة تهدف إلى تقليل تعقيدات التنفيذ. الرموز المقترحة مهيكلة بشكل شبكي بطريقة تقلل من الحد الأقصى لطول الأسلاك و هي كذلك قابلة للمقايسة. تمت نمذجة مفك رموز فتمك باستخدام لغة VHDL . النموذج الناتج هو عام الاستخدام و يستطيع فك أيٍّ من رموز فتمك. تم استخدام هذا النموذج مع عدة نماذج من رموز فتمك و تمت برهنة فاعلية رموز فتمك المقترحة و مقارنتها مع رموز عشوائية.

CHAPTER 1

INTRODUCTION

1.1 LDPC in brief

Low-density parity-check (LDPC) codes were first discovered by Robert Gallager [1] in the early 60s. For some reason, though, they were forgotten and the field lay dormant until the mid-90s when the codes were rediscovered by David MacKay and Radford Neal [2]. Since then, the class of codes has been shown to be remarkably powerful, comparable to the best known codes and performing very close to the theoretical limit of error-correcting codes.

The nature of the codes suggests a natural decoding algorithm operating directly on the parity check matrix. This algorithm has relatively low complexity and allows a high degree of parallelization when implemented in hardware, allowing high decoding speeds. The performance comes at a price, however. The memory requirements are very large, and the random nature of the codes leads to high interconnect complexities and routing congestions.

LDPC codes belong to linear block codes. An LDPC code is characterized by a very sparse parity-check matrix, **H**. For certain LDPC codes with very long block lengths, their BER performance has been found to be the closest to Shannon limit [3].

An LDPC code is defined by its parity-check matrix, **H** matrix. The structure of the **H** matrix is the main factor in determining the performance of the LDPC code. The **H** matrix can be constructed in two main approaches: randomly or deterministically.

There are many areas of research in this wide field of LDPC codes, like any other coding scheme. Some researches focuses on one area, but many combine more than one area. The research areas are:

- Code Design, where the **H** matrix, which characterizes the LDPC code, is designed to satisfy certain conditions and to achieve the maximum possible performance [4] [5].
- Decoding Algorithm, where the iterative decoding algorithm is modified in order to reduce the required computational power while retaining (almost) the same performance [6].
- Hardware Implementation, where the LDPC codec is implemented using different hardware platforms such as FPGA [7],[8], VLSI [9],[10],[11] and DSP [12]. This path of research is usually accompanied with the code design path. The emphasis here is targeted to the decoder design, which is the difficult part of LDPC, compared to the encoder part.
- Performance Analysis of different systems, where LDPC codes are used within different communication systems such as DSL [13], CDMA [14] and OFDM [15].
- Performance Analysis over different Channels, where the performance of LDPC codes is modeled for different channel such as AWGN channel [16] and [17], Rayleigh fading channel [14] and MIMO fading channel [18].

- Combination with other codes, like turbo codes [18], space-time codes [19] and convolutional codes [20].

There are many hardware implementations of the LDPC decoder in the literature. They differ in many aspects, such as architecture, platform, the type of code used in the testing, the number of iterations, the code length and the bit resolution. Hardware implementations can have different architectures. They can be parallel (as in [9]), serial (as in [21]) or can be mixed architecture (as in [7], [8], [10] and [11]). The decoder architecture can be implemented in different platforms, such as ASIC (as in [9], [10], [11] and [8]), FPGA (as in [7] and [8]) and DSP (as in [12]). The type of the code used in the hardware implementation will affect the resulting throughput and performance. Most of the hardware implementations found in the literature use random LDPC codes; regular (as in [10], [12] and [21]) and irregular (as in [8], [9] and [11]). However, some implementations use LDPC code designs that are intended to reduce hardware complexity and increase throughput (as in [7]). The results of LDPC decoder hardware implementations can also be affected by the number of iterations; which can go from 1(as in [10]) up to 64 (as in [9]). The hardware implementations differed also in the Block length. Some of them were as short as 200 (as in [12]) and some were as long as 9216 [7]. The number representation of the messages inside the decoder can be either fixed-point or floating-point. In the case of fixed-point representation, the bit resolution can vary from 4 bits (as in [9] and [10]) to 8 bits (as in [21]). In some studies, as in [12] , the floating-point representation is in fact fixed-point representation with 16 bit resolution.

1.2 Thesis Motivation

In order to utilize the LDPC codes developed in the literature and use them in practical systems, hardware implementations of LDPC encoder and decoder should be eventually realized. Although the implementation of the LDPC encoder is relatively easy, implementing the LDPC decoder is a more difficult task. The performance of an LDPC decoder depends on several factors. Among them are the length, the rate and the structure of the LDPC code. The complexity of an LDPC decoder depends on the configuration of the decoder architecture, the hardware platform used and the quantization parameters used in different processing elements of the decoder. The complexity is also dependent of the structure of the LDPC code structure.

This thesis is going to tackle different issues related to designing high-throughput LDPC decoder architectures. An LDPC code will be designed in such a way that the decoder architecture will require less wiring complexity. A fully-parallel LDPC decoder architecture will be designed, modeled in VHDL and implemented in FPGA, and its complexity will be demonstrated for different types of codes. Fixed-point simulation will be used to help determining optimum values of different parameters used in hardware implementation.

1.3 Thesis Contributions

This thesis has contributed the following:

1. A new LDPC code has been designed and has been shown to have lower complexity, compared to other random codes.

2. An LDPC decoder architecture has been designed with a generic structure that can work with different code rates, different bit resolutions and different maximum dynamic ranges.

3. Fixed-point performance of semi-random LDPC codes has been explored, and some hardware parameters have been selected.

1.4 Thesis Organization

The second chapter overviews LDPC codes with some depth. It covers LDPC's major components, encoder and decoder, with more emphasis on the decoder which is more complex and more critical for the performance. Two forms of the iterative belief-propagation algorithm are explained with some highlights on how to adapt them in hardware. Different areas of research in LDPC codes are briefed and two of them are chosen for further research in the next two chapters.

The third chapter studies and analyzes the quantization effects on the performance of LDPC codes. Fixed-point representation is used in simulation, and the effect of the number of bits representing bit node and check node message on the performance is discussed. The effect of the maximum value of the messages is considered as well. This fixed-point analysis is applied on semi-random LDPC codes, which is a special family of codes that has a simpler encoding process.

The fourth and fifth chapters discuss the design, performance and implementation of a new family of LDPC codes. In the fourth chapter, the motivation of the design is

explained and the structure of the code is analyzed to show the strong advantages that it has. The performance of the proposed code is shown in comparison to some other codes.

In the fifth chapter, the different decoder architecture approaches are discussed and compared showing their advantages and disadvantages. This comparison is supported by different examples. The structure of the decoder is then explained in three parts: the check node, the variable node and the interconnect network. Each part of the decoder is illustrated in diagrams and supported with its hardware model written in VHDL and synthesis results using Xilinx ISE software. The synthesis results will be shown for the proposed code and compared to some other codes.

The sixth chapter concludes with the major results of this thesis and discusses future research in this area and different methods to improve this work.

CHAPTER 2

ENCODING AND DECODING OF LDPC CODES

Like turbo codes, LDPC codes belong to the class of codes that can be decoded via an iterative decoding algorithm. The demonstration of capacity approaching performance in turbo codes stimulated interest in the improvement of Gallager's original LDPC codes to the extent that the performance of these two code types is now comparable in AWGN, and even some LDPC codes have been found to outperform turbo codes. The highly robust performance of LDPC codes in other types of channels such as partial-band jamming, quasi-static multi-input multi-output (MIMO) Rayleigh fading, fast MIMO Rayleigh fading, and periodic fading is demonstrated in [14] and [22] . Another advantage of LDPC codes is their inherent error-correction capability, as shown in Section 2.3.

In this chapter, the major components of an LDPC system will be introduced. LDPC encoding will be explained briefly. Then, different approaches of LDPC code design will be introduced and compared. After that, the decoding of LDPC will be explained with some depth. In the last section, the factors that affect the complexity of LDPC decoder will be discussed.□

2.1 Representation of LDPC Codes

LDPC can be represented in two different forms, in matrix form or in factor graph form.

2.1.1 Matrix Representation

Although LDPC codes can be generalized to non-binary alphabets, we shall consider only binary LDPC codes for the sake of simplicity. Because LDPC codes form a class of linear block codes, they may be described as a certain k-dimensional subspace C of the vector space $\mathbb{F}_2^n$ of binary n-tuples over the binary field $\mathbb{F}_2$. Given this, we may find a basis $B=\{g_0, g_1, \ldots, g_{k-1}\}$ which spans C so that each $c \in C$ may be written as $c=u_0g_0+u_1g_1+\ldots+u_{k-1}g_{k-1}$ for some $\{u_i\}$; more compactly, $c=uG$ where $u=[u_0\ u_1\ \ldots\ u_{k-1}]$ which is the information vector, and G is the so-called $k \times n$ generator matrix whose rows are the vectors $\{g_i\}$ (as is conventional in coding, all vectors are row vectors). The $(n-k)$-dimensional null space $C^{\perp}$ of G comprises all vectors $x \in \mathbb{F}_2^n$ for which $xG^T=0$ and is spanned by the basis $B^{\perp}=\{h_0, h_1, \ldots, h_{n-k-1}\}$. Thus, for each $c \in C$, $ch_i^T = 0$ for all i or, more compactly, $c\boldsymbol{H}^T=0$ where **H** is the so-called $(n-k)\times n$ parity-check matrix whose rows are the vectors $\{h_i\}$, and is the generator matrix for the null space $C^{\perp}$. The parity-check matrix **H** is so named because it performs $M=n-k$ separate parity checks on a received word.

A low-density parity-check code is a linear block code for which the parity-check matrix $\boldsymbol{H}$ has a low density of 1's and thus is a sparse matrix. A regular LDPC code is a linear block code whose parity-check matrix $\boldsymbol{H}$ contains exactly w_c 1's in each column and exactly $w_r=w_c(n/M)$ 1's in each row, where $w_r<<M$ (equivalently, $w_c<<n$). The code rate $R=k/n$ is related to these parameters via $R=1-w_c/w_r$ (this assumes $\boldsymbol{H}$ is full rank). If $\boldsymbol{H}$

is low density, but the number of 1's in each column or row is not constant, then the code is an irregular LDPC code. It is easier to see the sense in which an LDPC code is regular or irregular through its graphical representation.

2.1.2 Graphical Representation

Tanner considered LDPC codes (and a generalization) and showed how they may be represented effectively by a so-called bipartite graph (or factor graph), now called Tanner graph. The Tanner graph of an LDPC code is analogous to the trellis of a convolutional code in that it provides a complete representation of the code and it aids in the description of the decoding algorithm. A bipartite graph is a graph (nodes connected by edges) whose nodes may be separated into two types, and edges may only connect two nodes of different types. The two types of nodes in a Tanner graph are the variable nodes and the check nodes (which we shall call v-nodes and c-nodes, respectively). The Tanner graph of a code is drawn according to the following rule: check node j is connected to variable node i whenever element h_{ij} in $\boldsymbol{H}$ is a 1. One may deduce from this that there are M=n-k check nodes, one for each check equation, and n variable nodes, one for each code bit c_i. Further, the M rows of $\boldsymbol{H}$ specify the M c-node connections, and the n columns of $\boldsymbol{H}$ specify the n v-node connections.

As an example, consider a (10,5) linear block code with w_c=2 and w_r=w_c(n/M)=4 with the following $\boldsymbol{H}$ matrix

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Figure 2.1 a 5x10 H matrix

The Tanner graph corresponding to **H** is depicted in Figure 2.1. Observe that v-nodes c_0, c_1, c_2 and c_3 are connected to c-node f_0 in accordance with the fact that, in the zeroth row of **H**, $h_{00}=h_{01}=h_{02}=h_{03}=1$ (all others are zero). Observe that analogous situations hold for c-nodes f_1, f_2, f_3 and f_4 which correspond to rows 1, 2, 3 and 4 of ***H***, respectively. Note, as follows from the fact that $cH^T=0$, the bit values connected to the same check node must sum to zero. We may also proceed along columns to construct the Tanner graph. For example, note that v-node c_0 is connected to c-node f_0 and f_1 in accordance with the fact that, in the zeroth column of ***H***, $h_{00}=h_{10}=1$.

Note that the Tanner graph in this example is regular: each v-node has two edge connections and each c-node has four edge connections (that is, the degree of each v-node is 2 and the degree of each c-node is 4). This is in accordance with the fact that $w_c=2$ and $w_r=4$. It is also clear from this example that $mw_r=nw_c$.

For irregular LDPC codes, the parameters w_c and w_r are functions of the column and row numbers and so such notation is not generally adopted in this case. Instead, it is usual in the literature [23] to specify the v-node and c-node degree distribution polynomials, denoted by $\lambda(x)$ and $\rho(x)$, respectively. In the polynomial

$$\lambda(x) = \sum_{d=1}^{d_v} \lambda_d x^{d-1} \qquad (2.1)$$

λ_d denotes the fraction of all edges connected to degree-d v-nodes and d_v denotes the maximum v-node degree. Similarly, in the polynomial

$$\rho(x) = \sum_{d=1}^{d_c} \rho_d x^{d-1} \tag{2.2}$$

ρ_d denotes the fraction of all edges connected to degree-d c-nodes and d_c denotes the maximum c-node degree. Note for the regular code above, for which $w_c=d_v=2$ and $w_r=d_c=4$, we have $\lambda(x)=x$ and $\rho(x)=x^3$.

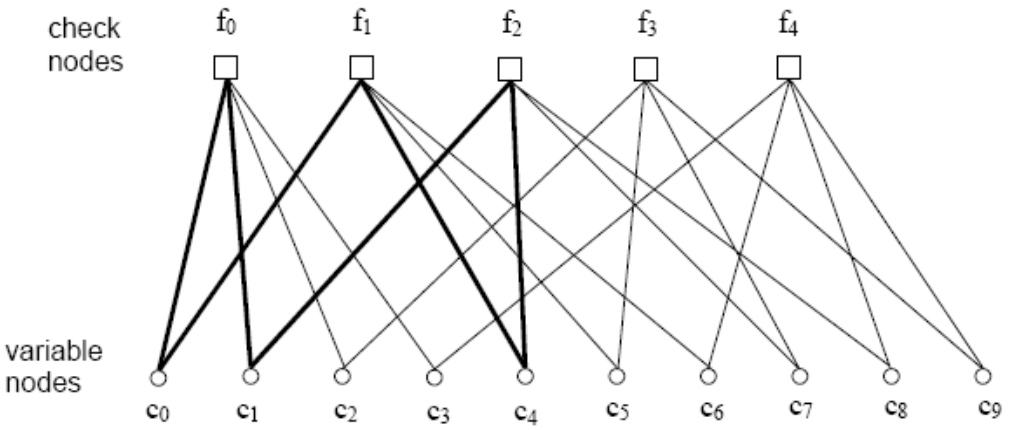

Figure 2.2 Tanner graph for the example code.

A cycle (or loop) of length v in a Tanner graph is a path comprising v edges which closes back on itself. The Tanner graph in the above example possesses a length-6 cycle as exemplified by the six bold edges in the figure. The girth γ of a Tanner graph is the minimum cycle length of the graph. The shortest possible cycle in a bipartite graph is clearly a length-4 cycle, and such cycles manifest themselves in the **H** matrix as four 1's that lie on the converse of a submatrix of **H**. We are interested in cycles, particularly

short cycles, because they degrade the performance of the iterative decoding algorithm used for LDPC codes. Eliminating short cycles have been proven to improve the performance of LDPC codes [24]. This fact will be made evident in the discussion of the iterative decoding algorithm.

2.2 Encoding of LDPC codes

Like any other linear block code, LDPC codes are encoded and decoded using two dual matrices, the generator matrix **G** and the parity-check matrix **H**. These two are related via the relation

$$GH^T = 0 \tag{2.3}$$

The parity-check matrix **H** can be written as,

$$H = [I \mid P] \tag{2.4}$$

where P is the parity check part and I is (n-k)×(n-k) identity matrix.

The generator matrix **G** can be written in the systematic form as,

$$G = [P^T \mid I] \tag{2.5}$$

where P^T represents the $k\times(n-k)$ transposed parity-check part and I represents $k\times k$ identity matrix.

If we consider a sequence of information bits $\boldsymbol{u}$ that contains k bits, the encoding process is simply multiplying this sequence by the generator matrix to get

$$c = uG \tag{2.6}$$

G and **H** can be found from each other using Gaussian elimination.

Some LDPC codes are designed to have simpler encoding schemes, such as lower-triangular shape encoding, low-density generator matrices [25], the cyclic parity-check matrices [5] and the semi-random LDPC codes[26]. The last one, in particular, will be studied in more detail in Chapter 3.

2.3 Iterative Decoding Algorithm

2.3.1 Overview

In addition to introducing LDPC codes in his Ph.D dessertation in 1960 [1], Gallager also provided a decoding algorithm that is typically near optimal. Since that time, other researchers have independently discovered that algorithm and related algorithms, although sometimes for different applications [2], [27]. The algorithm iteratively computes the distributions of variables in graph-based models and comes under different names, depending on the context. These names include: the sum-product algorithm (SPA), the belief propagation algorithm (BPA), and the message passing algorithm (MPA). The term "message passing" usually refers to all such iterative algorithms, including the SPA (BPA) and its approximations.

Much like optimal (maximum a posteriori, MAP) symbol-by-symbol decoding of trellis codes, we are interested in computing the a posteriori probability (APP) that a given bit in the transmitted codeword $\boldsymbol{c}=[c_0\ c_1\ \ldots.\ c_{n-1}]$ equals 1, given the received word $\boldsymbol{y}=[y_0\ y_1\ \ldots.\ y_{n-1}]$. Without loss of generality, let us focus on the decoding of bit c_i so that we are interested in computing the APP

$$\Pr(c_i = 1 | y) \tag{2.7}$$

Or the APP ratio (also called the likelihood ratio, LR)

$$l(c_i) = \frac{\Pr(c_i = 0 \mid y)}{\Pr(c_i = 1 \mid y)} \tag{2.8}$$

Later we will extend this to the more numerically stable computation of the log-APP ratio, also called the log-likelihood ratio (LLR):

$$L(c_i) = \log\left(\frac{\Pr(c_i = 0 \mid y)}{\Pr(c_i = 1 \mid y)}\right) \tag{2.9}$$

where here and in the forthcoming parts, the natural logarithm is assumed.

The MPA for the computation of $Pr(c_i = 1|y)$, $l(c_i)$, or $L(c_i)$ is an iterative algorithm which is based on the code's Tanner graph. Specifically, we imagine that the v-nodes represent the processors of one type, c-nodes represent processors of another type, and the edges represent message paths. In one half iteration, each v-node processes its input messages and passes its resulting output messages up to neighboring c-nodes (two nodes are said to be neighbors if they are connected by an edge). This is depicted in Figure 2.3 for the message $m_{\uparrow 02}$ from v-node c_0 to c-node f_2 (the subscripted arrow indicates the direction of the message, keeping in mind that our Tanner graph convention places c-nodes above v-nodes). The information passed concerns Pr(c_0=b | input messages), $b \in \{0,1\}$, the ratio of such probabilities, or the logarithm of the ratio of such probabilities. Note in the figure that the information passed to c-node f_2 is all the information available to v-node c_0 from the channel and through its neighbors, excluding

c-node f_2; that is, only extrinsic information is passed. Such extrinsic information $m_{\uparrow ij}$ is computed for each connected v-node/c-node pair c_i/f_i at each half-iteration.

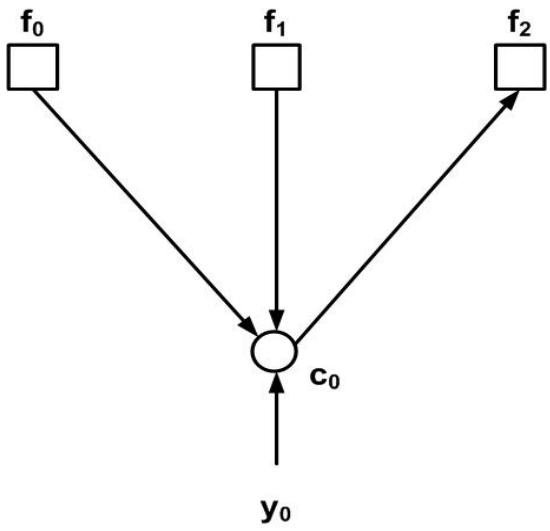

Figure 2.3 Message passing from variable node c0 to check node f2.

In the other half iteration, each c-node processes its input messages and passes its resulting output messages down to its neighboring v-nodes. This is depicted in Figure 2.4 for the message $M_{\downarrow 03}$ from c-node f_0 down to v-node c_3. The information passed concerns Pr(check equation f_0 is satisfied | input message), $b \in \{0,1\}$, the ratio of such probabilities, or the logarithm of the ratio of such probabilities. Note, as for the previous case, only extrinsic information is passed to v-node c_3. Such extrinsic information $m_{\uparrow ji}$ is computed for each connected c-node/v-node pair f_i/c_i at each half-iteration.

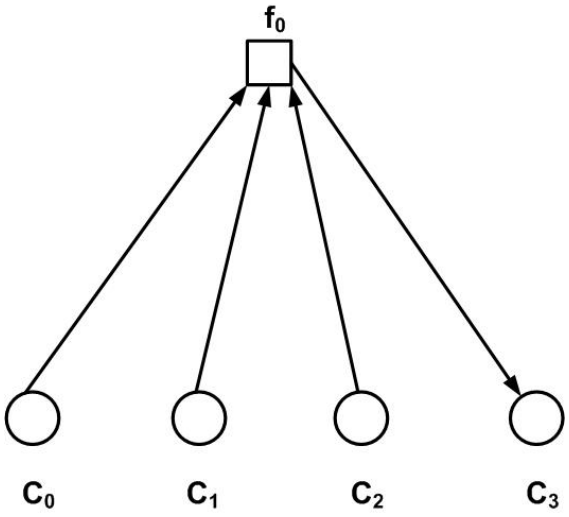

Figure 2.4 Message passing from check node f0 to variable node c4.

After a prescribed maximum number of iterations or after some stopping criterion has been met, the decoder computes the APP, the LR, or the LLR from which decisions on the bits c are made. One example stopping criterion is to stop iterating when $\hat{c}H^T = 0$, where $\hat{c}$ is a tentatively decoded codeword.

The MPA assumes that the messages passed are statistically independent throughout the decoding process. When the y_i are independent, this independence assumption would hold true if the Tanner graph possessed no cycles. Further, the MPA would yield exact APPs (or LRs or LLRs, depending on the version of the algorithm) in this case. However, for a graph of girth γ, the independence assumption is only true up to the $\gamma/2$-th iteration, after which messages start to loop back on themselves in the graph's various cycles. Still, simulations have shown that the message passing algorithm is generally very effective provided length-four cycles are avoided. Lin *et al*. [28] showed that some configurations of length-four cycles are not harmful. It was shown in [29] how the message-passing

schedule described above and below (the so-called flooding schedule) may be modified to mitigate the negative effects of short cycles.

In the following, the "probability domain" version of the SPA (which computes APPs) and its log-domain version, the log-SPA (which computes LLRs), will be presented as well as certain approximations.

2.3.2 Probability-Domain SPA Decoder

We start by introducing the following notation:

- V_j={v-nodes connected to c-node f_j }
- $V_{j\backslash i}$= {v-nodes connected to c-nodes f_j }\ {v-node c_i}
- C_i ={c-nodes connected to v-node c_i }
- $C_{i\backslash j}$={c-nodes connected to v-nodes c_i }\ {c-node f_j }
- $M_v(\sim i)$={messages from all v-nodes except node c_i}
- $M_c(\sim j)$={messages from all c-nodes except node f_j }
- $P_i = Pr(c_i = 1 \mid y_i)$
- S_i = event that check equations involving c_i are satisfied
- , $qij(b)=Pr(c_i=b|S_i,y_i,M_c(\sim j))$, where $b \in \{0,1\}$. For the APP algorithm presently under consideration , $m_{\downarrow ij} = q_{ij}(b)$; for the LR algorithm, $m_{\downarrow ij} = q_{ij}(0)/q_{ij}(1)$; and for the LLR algorithm $m_{\downarrow ij} = log[q_{ij}(0)/q_{ij}(1)]$.

- $r_{ji}(0)$= Pr(check equation f_j is satisfied | $c_i=b$, $M_v(\sim i)$), where $b \in \{0,1\}$. For the APP algorithm presently under consideration, $m_{\downarrow ji} = r_{ji}(b)$; for the LR algorithm, $m_{\downarrow ji} = r_{ji}(0)/r_{ji}(1)$; and for the LLR algorithm, $m_{\downarrow ji} = log[r_{ji}(0) / r_{ji}(1)]$

Note that the messages $q_{ij}(b)$, while interpreted as probabilities here, are random variables (rv's) as they are functions of the rv's y_i and other messages which are themselves rv's. Similarly by virtue of the message passing algorithm, the message $r_{ji}(b)$ are rv's.

Consider now the form of $q_{ij}(0)$ which, given our new notation and the independence assumptions, we may express as (see Figure 2.5)

$$\begin{aligned} q_{ij}(0) &= Pr(c_i=0 | y_i, S_i, M_c(\sim j)) \\ &= (1-P_i) Pr(S_i | c_i=0, y_i M_c(\sim j)) / Pr(S_i) \\ &= K_{ij}(1-P_i) \prod_{j' \in C_i \setminus j} r_{j'i}(0) \end{aligned} \tag{2.10}$$

where we used Bayes' rule twice to obtain the second line and the independence assumption to obtain the third line. Similarly,

$$q_j(1) = K_{ij} P_i \prod_{j' \in C_i \setminus j} r_{j'i}(1) \tag{2.11}$$

The constants K_{ij} are chosen to ensure that $q_{ij}(0)+q_{ij}(1)=1$.

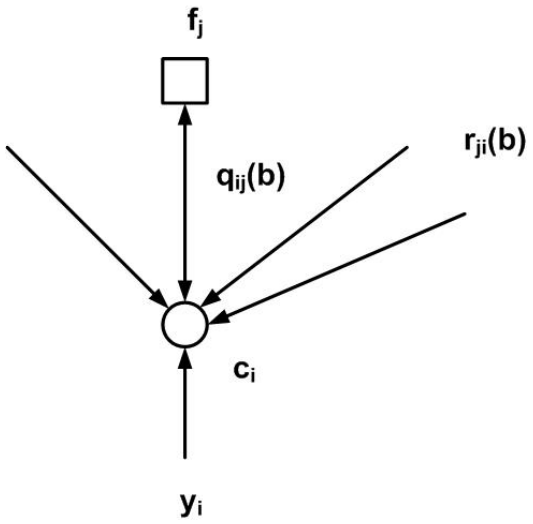

Figure 2.5 Message Passing Half-Iteration For The Computation of qij(b).

To develop an expression for the r_{ji}(b), we need the following result:

Result 1. [30] consider a sequence of M independent binary digits a_i for which $\Pr(a_i=1)=p_i$. Then, the probability that $\{a_i\}_{i=1}^{M}$ contains an even number of 1's is

$$\frac{1}{2}+\frac{1}{2}\prod_{l=i}^{M}(1-2p_i) \tag{2.12}$$

In view of this result together with the correspondence $p_i \leftrightarrow q_{ij}(1)$, we have (see Figure 2.6)

$$r_{ji}(0)=\frac{1}{2}+\frac{1}{2}\prod_{i'\in V_j\setminus i}(1-2q_{ij}(1)) \tag{2.13}$$

Since, when $c_i=0$, the bits $\{c_{i'}: i'\in V_{j\setminus i}\}$ must contain an even number of 1's in order for check equation f_j to be satisfied. Clearly,

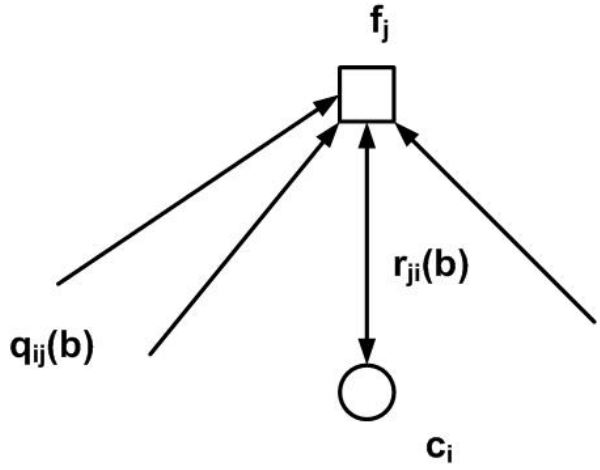

Figure 2.6 Message passing half-iteration for the computation of $r_{ji}(b)$.

$$r_{ji}(1) = 1 - r_{ji}(0) \tag{2.14}$$

The MPA for the computation of the APPs is initialized by setting $q_{ij}(b)=Pr(c_i= b| y_i)$ for all i , j for which $h_{ij}=1$ (that is , $q_{ij}(1)= P_i$ and $q_{ij}(0)=1-P_i$). Here, y_i represents channel symbol that was actually received (i.e., it is not a variable here). We consider the following special cases.

The binary erasure channel *(BEC)*. In this case, $y_i \in \{0,1,E\}$ where E is the erasure symbol, and we define $\delta = \Pr(y_i = E \mid c_i = b)$ to be the erasure probability. Then it is easy to see that

$$\Pr(c_i = b \mid y_i) = \begin{cases} 1 & when \;\; y_i = b \\ 0 & when \;\; y_i = b^c \\ 1/2 & when \;\; y_i = E \end{cases} \tag{2.15}$$

where b^c represents the complement of b.

The binary symmetric channel *(BSC)*. In this case, $y_i \in \{0,1\}$ and we define $\varepsilon = Pr(y_i = b^c \mid c_i = b)$ to be the error probability. Then it is obvious that

$$\Pr(c_i = b \mid y_i) = \begin{cases} 1-\varepsilon & when\ y_i = b \\ \varepsilon & when\ y_i = b^c \end{cases} \tag{2.16}$$

The binary-input AWGN channel *(BI-AWGNC)*. We first let $x_i = 1-2c_i$ be the i-th transmitted binary value; note $x_i = +1(-1)$ when $c_i = 0(1)$. We shall use x_i and c_i interchangeably. Then, the i-th received sample is $y_i = x_i + n_i$, where the n_i are independent and normally distributed as $\eta(0,\sigma^2)$. Then it is easy to show that

$$Pr(x_i = x \mid y_i) = \left[1 + exp(-2yx / s^2)\right]^{-1} \tag{2.17}$$

where $x \in \{\pm 1\}$.

Summary of the Probability-Domain SPA Decoder

1. For $i = 0, 1, \ldots, n-1$, set $Pi = Pr(ci = 1 \mid y_i)$ where y_i is the i-th received channel symbol. Then set $q_{ij}(0) = 1-Pi$ and $q_{ij}(1) = P_i$ for all i, j for which $hij = 1$.

2. Update $\{r_{ji}(b)\}$ using equations (2.13) (2.14) .

3. Update $\{ q_{ji}(b) \}$ using equations (2.10) and (2.11). Solve for the constants K_{ij}.

4. For $i = 0, 1, \ldots, n-1$, compute

$$Q_i(0) = K_i(1 - P_i) \prod_{j \hat{I} C_i} r_{ji}(0) \tag{2.18}$$

and

$$Q_i(1) = K_i P_i \prod_{j \hat{I} C_i} r_{ji}(1) \tag{2.19}$$

where the constants K_i are chosen to ensure that *Qi(0) + Qi(1) = 1*.

5. For $i = 0, 1, \ldots, n\text{-}1$, set

$$\hat{c}_i = \begin{cases} 1 & \text{if } Q_i(1) > Q_i(0) \\ 0 & \text{else} \end{cases} \tag{2.20}$$

If $\hat{c}H^T = 0$ or the number of iteration equals the maximum limit, stop; else, go to step 2.

If the stopping criterion (in step 5) is chosen to be computing and verifying $\hat{c}H^T = 0$, the iterative algorithm will provide us with an error-detection mechanism that will detect an uncorrected codeword with certainty.

2.3.3 Log-Domain SPA Decoder

As with the probability-domain Viterbi and BCJR (Bahl-Cocke-Jelinek-Raviv) algorithms, the probability-domain SPA suffers because multiplications are involved (additions are less costly to implement) and many multiplications of probabilities are involved, which could become numerically unstable. Thus, as with the Viterbi and BCJR algorithms, a log-domain version of the SPA is to be preferred. To do so, we first define the following LLRs:

$$L(c_i) = \log\left(\frac{\Pr(c_i = 0 \mid y_i)}{\Pr(c_i = 1 \mid y_i)}\right) \tag{2.21}$$

$$L(r_{ji}) = \log\left(\frac{r_{ji}(0)}{r_{ji}(1)}\right) \tag{2.22}$$

$$L(q_{ij}) = \log\left(\frac{q_{ij}(0)}{q_{ij}(1)}\right) \tag{2.23}$$

$$L(Q_i) = \log\left(\frac{Q_i(0)}{Q_i(1)}\right) \tag{2.24}$$

The initialization steps for the three channels under consideration thus become:

$$L(q_{ij}) = L(c_i) = \begin{cases} +\infty, & y_i = 0 \\ -\infty, & y_i = 1 \\ 0, & y_i = E \end{cases} \qquad (BEC) \tag{2.25}$$

$$L(q_{ij}) = L(c_i) = (-1)^{y_i} \log\left(\frac{1-\varepsilon}{\varepsilon}\right) \quad \text{(BSC)} \tag{2.26}$$

$$L(q_{ij}) = L(c_i) = 2y_i / \sigma^2 \quad \text{(BI-AWGNC)} \tag{2.27}$$

For step 1, we first replace $r_{ji}(0)$ with $1\text{-}rji(1)$ in () and rearrange it to obtain

$$1 - 2r_{ji}(1) = \prod_{i' \hat{I} V_j / i} (1 - 2q_{ij}(1)) \tag{2.28}$$

Now using the fact that $tanh\left[\frac{1}{2}log(p_0 / p_1)\right] = p_0 - p_1 = 1 - 2p_1$, we may rewrite the equation above as

$$tanh\left(\frac{1}{2}L(r_{ji})\right) = \prod_{i' \hat{I} V_j \backslash i} tanh\left(\frac{1}{2}L(q_{ij})\right) \tag{2.29}$$

The problem with these expressions is that we are still left with a product and the complex tanh function. We can solve this as follows. First, factor $L(q_{ij})$ into its sign and magnitude:

$$L(q_{ij}) = \alpha_{ij}\beta_{ij} \tag{2.30}$$

$$\alpha_{ij} = sign[L(q_{ij})] \tag{2.31}$$

$$\beta_{ij} = \left| L(q_{ij}) \right| \tag{2.32}$$

so that (2.29)may be rewritten as

$$\tanh\left(\frac{1}{2}L(r_{ji})\right) = \prod_{i' \in V_j \setminus i} \alpha_{i'j} . \prod_{i' \in V_j \setminus i} \tanh\left(\frac{1}{2}\beta_{i'j}\right) \tag{2.33}$$

we then have

$$L(r_{ji}) = \prod_{i'} \alpha_{i'j} . \, 2\tanh^{-1}\left(\prod_{i'} \tanh\left(\frac{1}{2}\beta_{i'j}\right)\right) \tag{2.34}$$

$$= \prod_{i'} \alpha_{i'j} . \, 2\tanh^{-1}\log^{-1}\log\left(\prod_{i'} \tanh\left(\frac{1}{2}\beta_{i'j}\right)\right) \tag{2.35}$$

$$= \prod_{i'} \alpha_{i'j} . \, 2\tanh^{-1}\log^{-1}\sum_{i'} \log\left(\tanh\left(\frac{1}{2}\beta_{i'j}\right)\right) \tag{2.36}$$

$$= \prod_{i' \in V_j \setminus i} \alpha_{i'j} . \, \phi\left(\sum_{i' \in V_j \setminus i} \phi(\beta_{i'j})\right) \tag{2.37}$$

where we have defined

$$j(x) = -\log[\tanh(x / 2)] = \log\left(\frac{e^x + 1}{e^x - 1}\right) \tag{2.38}$$

and used the fact that $\phi^{-1}(x) = \phi(x)$ when $x>0$. The function is fairly well behaved, as shown in Figure 2.7, and so may be implemented by a look-up table.

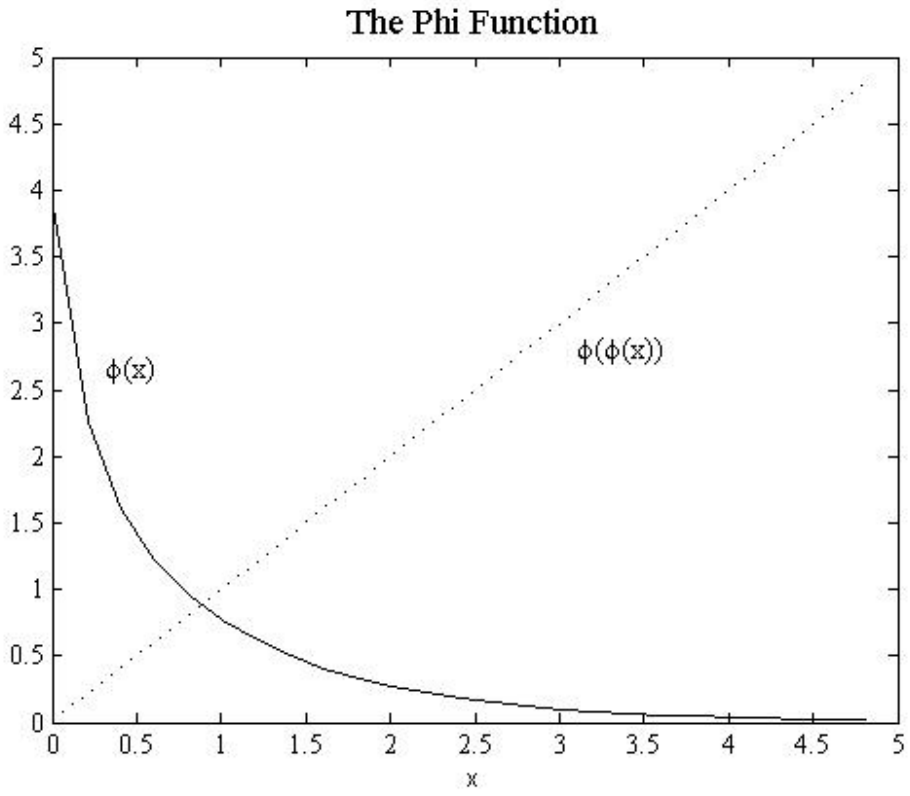

Figure 2.7 Plot of the Ø(x) function.

For step 2, we simply divide equation (2.10) by (2.11) and take the logarithm of both sides to obtain

$$L(q_{ij}) = L(c_i) + \sum_{j' \in C_i \setminus j} L(r_{j'i}) \tag{2.39}$$

Step 3 is similarly modified so that

$$L(Q_i) = L(c_i) + \sum_{j \in C_i} L(r_{ji}) \tag{2.40}$$

Summary of the Log-Domain SPA Decoder

1. For $i = 0, 1, \ldots, n\text{-}1$, initialize $L(q_{ij})$ according to (2.25) for all i,j for which $h_{ij} = 1$.

$$L(q_{ij}) = L(c_i) = 2y_i / \sigma^2 \quad \text{(BI-AWGNC)}$$

2. Update $\{L(rji)\}$ using equation (2.37).

$$= \prod_{i' \in V_j \setminus i} \alpha_{i'j} \cdot \phi\left(\sum_{i' \in V_j \setminus i} \phi(\beta_{i'j}) \right) \tag{2.41}$$

3. Update $\{L(qij)\}$ using equation (2.39)

$$L(q_{ij}) = L(c_i) + \sum_{j' \in C_i \setminus j} L(r_{j'i}) \tag{2.42}$$

4. Update $\{L(Qi)\}$ using equation (2.40).

$$L(Q_i) = L(c_i) + \sum_{j \in C_i} L(r_{ji}) \tag{2.43}$$

5. For $i = 0, 1, \ldots, n\text{-}1$, set

$$\hat{c}_i = \begin{cases} 1 & \text{if } L(Q_i) < 0 \\ 0 & \text{else} \end{cases} \tag{2.44}$$

If $\hat{c}H^T = 0$ or the number of iterations equals the maximum limit, stop; otherwise, go to step 2.

2.3.4 LDPC Decoder Complexity

The complexity of the decoding process can be divided into three parts: the complexity of the check and variable processing nodes, the complexity of the memory management process and the complexity of the interconnect network through which the messages are passed.

When dealing with the issue of implementing LDPC decoders, three types of complexities are encountered:

1. Complexity of node computations (check node and variable node processing).
2. Complexity of the interconnecting network through which the messages are exchanged.
3. Complexity of the number of times the computations need to be repeated (number of iterations)

Designing a hardware LDPC decoder is a tradeoff process between these three factors. In Chapter 4, an LDPC code will be designed such that the interconnect network complexity is minimized. The design approach will be based on using deterministic methods, rather than random, to distribute the check nodes and the variable nodes into a mesh of clusters. Those clusters will be connected to each other only if they are close enough to each other. This approach will eliminate the existence of long connections, or edges, that usually cause more delay to the system, and therefore less throughput. In Chapter 5, the implementation of this LDPC code will be done in a fully parallel fashion in order to reduce the memory management requirements. Reducing the complexity of any, or all, the above mentioned factors will improve the throughput of the decoder. Another factor that can also improve the throughput is the number of iterations before aborting the iterative decoding algorithm. As the number of iterations decreases, the throughput will increase, but with the expense of losing some performance.

CHAPTER 3

FINITE-PRECISION PERFORMANCE OF SEMI-RANDOM LDPC CODES

3.1 Introduction

When designing LDPC codes, the performance is usually measured through floating-point simulations. The floating-point simulation requires less time to encode in software and gives better results. However, when considering hardware implementation, using floating-point representation is not practical and some times, is not even realizable. Fixed-point representation is used instead. Hardware implementation deals with data in finite precision rather than infinite precision. The process of deciding the accuracy of the finite precision representation and, hence, setting the parameters of the fixed-point simulation, is a tradeoff process. Instead of selecting these parameters in a late stage of the hardware simulation, they can be selected earlier in the stage of software simulations using fixed-point simulation approach. Different hardware parameters can be included in the simulation process, such as the number of bits representing each message value and the dynamic range of these values and the maximum number of iterations used by the decoding algorithm.

In this chapter, we consider a specific family of LDPC codes known as semi-random LDPC codes [26]. This family of codes has a major advantage among others; it has a much simpler and faster encoding process. Finding the generator matrix of an LDPC code usually requires performing Gaussian elimination on the systematic form of the parity-check matrix. This implies more requirements in computations and memory. In semi-random LDPC codes, the parity-check matrix is directly used for the encoding process which will save memory and hardware area. This simplified encoding process will improve throughput and will reduce hardware complexity requirements [26].

3.2 Structure of Semi-Random LDPC Codes

Like any other linear block code, LDPC codes can be encoded by directly multiplying the message vector with the generator matrix **G**:

$$c = m * G \qquad (3.1)$$

$$c=M*G \qquad (3.2)$$

G is generally not sparse, which will increase the complexity of the encoding process.

Some codes can have a generator matrix **G** with a special structure that will hugely reduce the complexity of the encoding process. An example of such code structure is semi-random LDPC codes.

In semi-random LDPC codes [26], part of **H** is random and the other is deterministic.

$$\boldsymbol{H}=[H^p,H^d] \qquad (3.3)$$

$\mathbf{H}^p$ is a square matrix

$$H^{p} = \begin{bmatrix} 1 & 0 & & & & & 0 \\ 1 & 1 & & & & & \\ & 1 & . & & & & \\ & & . & . & & & \\ & & & . & . & & \\ & & & & . & . & \\ & & & & & . & 1 \\ 0 & & & & & 1 & 1 \end{bmatrix}_{(n-k)\times(n-k)} \qquad (3.4)$$

to construct $\mathbf{H}^{d}$, we select a number t such that t divides n-k, and n-k divides kt

$$H^{d} = \begin{bmatrix} H^{d1} \\ . \\ . \\ . \\ H^{dt} \end{bmatrix}_{(n-k)\times k}, \left[H^{di}\right]_{(\frac{n-k}{t})\times k} \qquad (3.5)$$

In each H^{di}, $i=1, \ldots, t$, we randomly create one element 1 per column , and $(kt/n-k)$ 1s per row. The resultant $\boldsymbol{H}^{di}$ has a column weight of t and a row weight of kt/n-k.

$\boldsymbol{p}=\{p_i\}$ can be calculated from given $d=\{d_{ij}\}$ as

$$p_1 = \sum h_{ij}^{d} d_j \ , \quad p_i = p_{i-1} + \sum h_{ij}^{d} d_j \qquad (3.6)$$

The way of calculating the parity bits will significantly reduce the complexity of the encoder. This family of semi-random LDPC codes has less complex encoding process, with essentially the same performance as randomly-constructed LDPC codes [26].

3.3 Fixed-Point Implementation of Semi-Random LDPC Codes

Floating-point simulation results of semi-random LDPC codes have shown good performance over Additive White Gaussian Noise channel (AWGN) [26]. However, since hardware implementation is usually costly, the hardware performance of semi-random LDPC codes can be estimated by performing fixed-point simulation. The fixed-point simulation will investigate the effect of quantizing the messages of the iterative decoding algorithm on the performance of the system. More accurately, the effect of quantization will be evaluated when varying any of three main parameters of the decoding algorithm: the maximum dynamic range of message values, the number of bits representing each value and the maximum number of iterations in the decoding algorithm.

3.4 Simulation Results

In this work, we investigate the performance of semi-random LDPC codes when numbers are represented in fixed-point format, instead of floating-point format, in order to simulate real hardware. The effect of fixed-point implementation on the performance will be measured by varying different parameters. Such investigation will constitute a midway step in checking the validity of this family of codes for real hardware implementation. Also, such an investigation will give an insight into possible tradeoffs that can be done to achieve certain throughput or hardware complexity requirements.

What usually happens in software simulations is that the values of the messages inside the iterative decoding algorithm are represented in floating-point format. In this way, the

message values are passed accurately between nodes without losing any information. On the other hand, real hardware can hardly work with floating-point representation. In most of the cases, it deals with fixed-point representation. So, the exchanged messages in the iterative decoding algorithm will be represented in fixed-point format and thus their values will be rounded in each iteration causing some loss of accuracy. This rounding of values will be done according to the following procedure:

- If |message value|>M-Δ , where $\Delta = M / 2^{nbit-1}$

  then message value= (M-Δ) × sign(message value)

- Else, $$\text{Message Value} = \left\lfloor \frac{\text{Message Value}}{\frac{M}{2^{nbits-1}}} + 0.5 \right\rfloor \times \frac{M}{2^{nbits-1}} \qquad (3.7)$$

This equation is illustrated in Figure 3.1

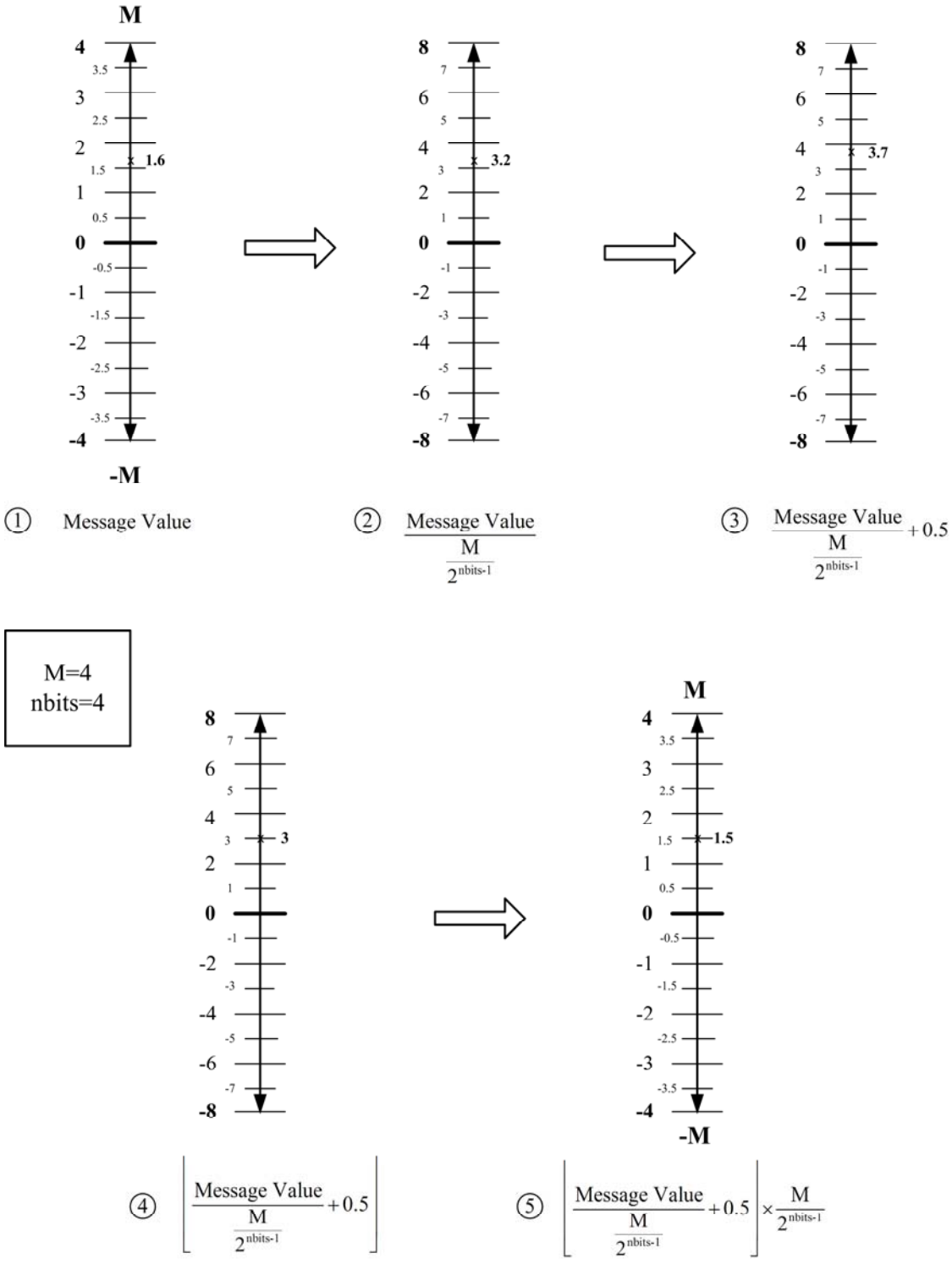

Figure 3.1 Illustration of the quantization procedure.

The accumulated loss of information in each iteration will affect the bit error rate (FER) performance of the system. The effect of fixed-point implementation on the performance will be measured by varying three parameters: the maximum dynamic range of message values M, the number of bits representing each value (The bit precision) *nbit* and the maximum number of iteration in the decoding algorithm It_{max}.

3.4.1 Effect of Varying Dynamic Range *M*

In Figure 3.2, the simulation is done for various values of dynamic range M. We notice that the curves representing the ±4 & ±4.5 are the closest to the floating-point curve, while the ±6.5, ±6 and ±3.5 are the farthest from the floating-point curve. The ±5 curve gets closer to the floating-point curve after SNR =3.25 dB. So, the range between ±4 till ±5 is an appropriate choice for the dynamic range.

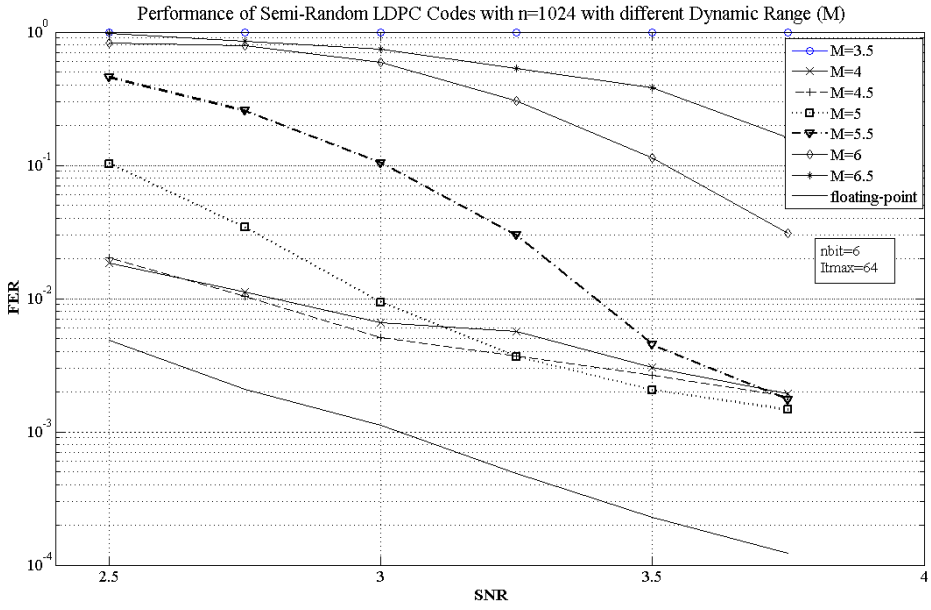

Figure 3.2 Performance for different dynamic ranges *M*

Another plot that will help in selecting the value of the dynamic range is Figure 3.3. In this figure, the FER is deteriorating when the dynamic range is out of the range 3.6→5.

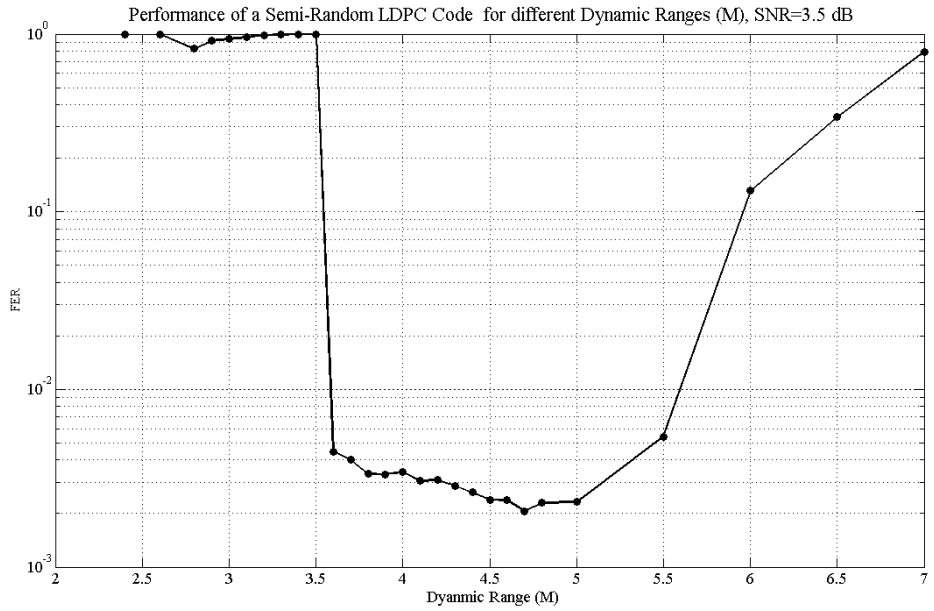

Figure 3.3 Performance for different values of *M* at SNR=3.5

Figure 3.4 shows the histogram of LLR messages of 10 decoded codewords during all the iterations of the decoding process. The histogram shows that the vast majority of message values are below 10. When assessing this histogram statistically, only 1.52% of the messages have values greater that 10, only 4% of the messages have values greater than 8 and 16% of the messages have values greater than 5.5.

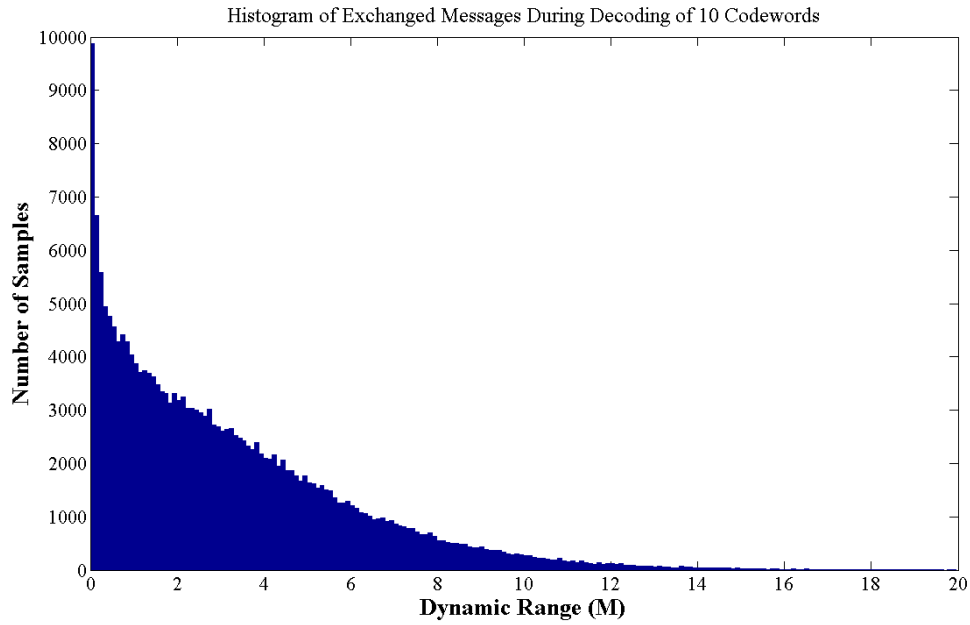

Figure 3.4 Histogram of LLR messages.

Although this argument may suggest using as higher value of dynamic range (M) as possible in order to avoid clipping the message values, the previous graph, Figure 3.3, suggests a limited range of M in order to get optimal performance. This apparent contradiction may be explained by discussing the change of both *nbit* and M. Increasing the value of M will reduce the number of messages whose values are clipped, and thus reducing the quantization errors. On the other hand, increasing the value of M, for a fixed *nbit*, will increase the width between quantization resulting in more quantization errors. For example, if we have a fixed value for *nbit* and if we select the value of M to be 5.5, the message samples with values less than 5.5 are going to have less quantization errors because the distance between levels has decreased. However, for the 16% of the message samples that are higher than 5.5 will have more quantization errors because of the clipping. So, selecting an appropriate value of M should be done in conjunction with *nbit*. This is going to be explored further in Section 3.4.4.

3.4.2 Effect of Varying number of iterations It_{max}

In Figure 3.5, we notice a significant improvement when changing It_{max} from 10 to 100. When going from It_{max} =10 to 20 (at FER=2×10^{-2}), the performance improves by 0.7 dB. When going from It_{max} =20 to 30 (at FER=2×10^{-2}), the performance improves by 0.1 dB. When going from It_{max} =30 to 60 (at FER=2×10^{-2}), the performance improves by 0.08 dB. When going from It_{max} =60 to 100 (at FER=1×10^{-2}), the performance will remain almost the same. It should be noticed that increasing It_{max} by a factor of 2 doesn't necessarily double the total delay (or the frame's processing time) because not all frames will need to take It_{max} iterations.

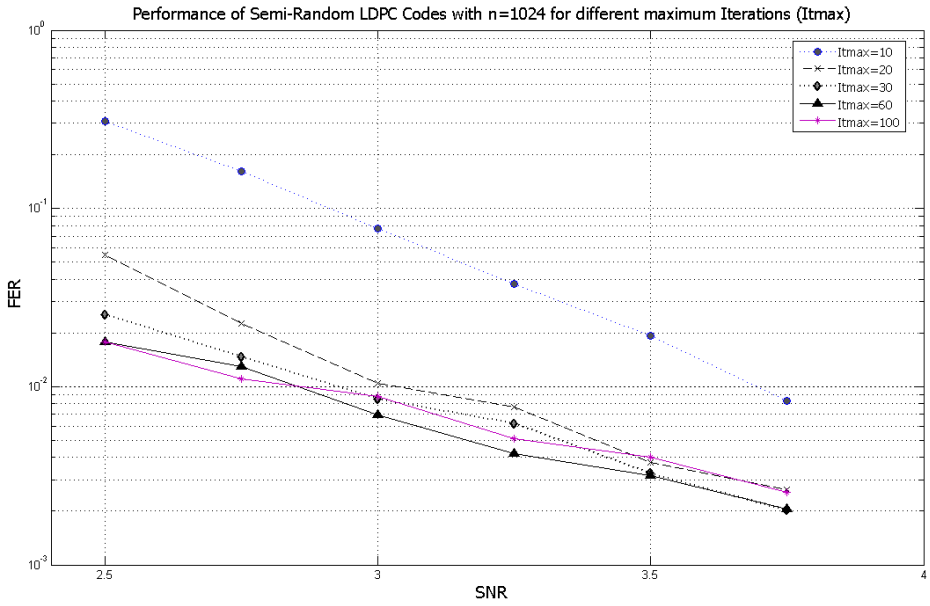

Figure 3.5 Performance for different maximum iterations It_{max}.

3.4.3 Effect of Varying number of bits

The number of bits used to represent LLRs is very important in the hardware implementation of an LDPC decoder. It will affect speed, performance and complexity.

Figure 3.6 shows the simulation of a (1024,512) Semi-Random LDPC code for different bit resolutions. The dynamic range is set to 4 and the maximum number of iterations is selected to be 64. The bit precision is varied from 3 to 12. The performance improves always with increasing the number of resolution bits *nbit*.

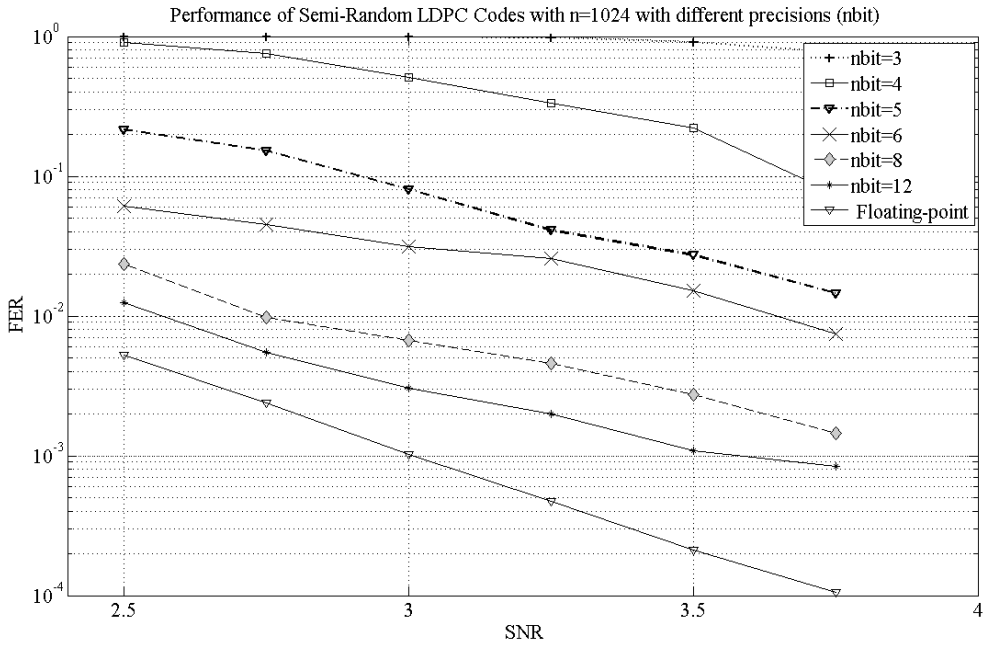

Figure 3.6 Performance for different bit precisions *nbit*.

3.4.4 Joint Optimization of Dynamic Range *M* and Resolution Bits *nbit*

In order to be more accurate in selecting the values of *nbit* and *M*, the performance should be investigated as a function of both variables. Figure 3.7 and Figure 3.8 show the 3D view of the FER performance of a (512, 1024) semi random LDPC code as a function of *M* and *nbit*. The previous results discussed in sections 3.1 and 3.2 are still valid but they are limited to special cases.

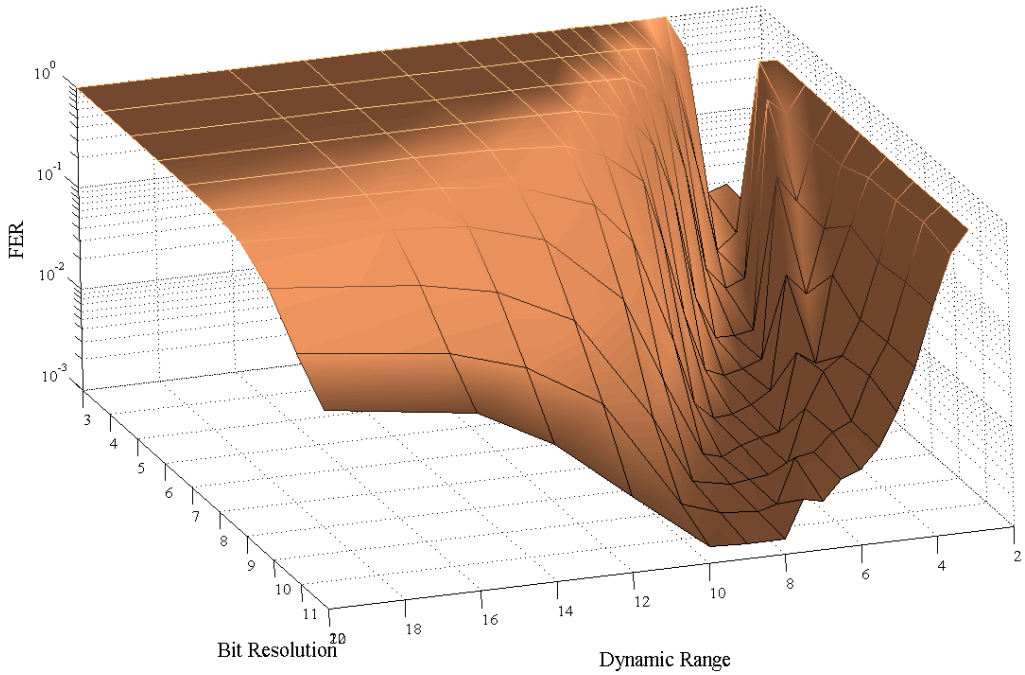

Figure 3.7 3D view of the performance as a function of *nbit* and *M*

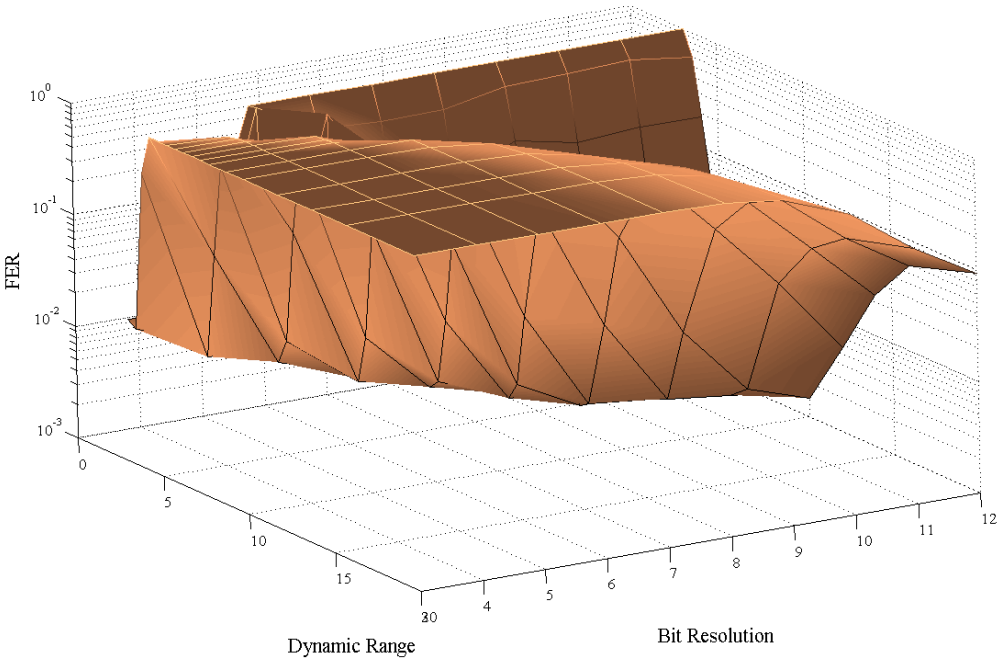

Figure 3.8 Another 3D view of the performance as a function of *nbit* and *M*

TABLE 3.1 Data of the 3D plots

n-bit	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
M										
3.0	1.9E-02	1.7E-02	9.7E-01	9.7E-01	9.5E-01	9.8E-01	9.9E-01	9.9E-01	9.9E-01	9.9E-01
3.5	1.6E-02	1.1E-02	1.0E+00	5.2E-01	4.9E-01	5.0E-01	6.4E-01	6.6E-01	7.0E-01	5.9E-01
4.0	4.4E-01	**8.0E-03**	**7.5E-03**	7.6E-01	1.3E-01	1.2E-01	2.3E-01	1.7E-01	1.9E-01	1.9E-01
4.5	1.0E+00	7.0E-02	**6.2E-03**	**4.2E-03**	2.4E-01	2.9E-02	7.7E-02	4.9E-02	5.1E-02	5.0E-02
5.0	1.0E+00	4.8E-01	1.0E-02	**3.9E-03**	3.9E-03	4.6E-02	1.1E-02	2.3E-02	2.5E-02	1.9E-02
5.5	1.0E+00	9.4E-01	1.0E-01	**3.9E-03**	**3.3E-03**	3.5E-03	1.5E-02	1.6E-02	9.4E-03	8.9E-03
6.0	1.0E+00	9.9E-01	5.0E-01	1.1E-02	**3.0E-03**	**2.7E-03**	2.5E-03	7.4E-03	7.6E-03	5.6E-03
6.5	1.0E+00	1.0E+00	7.6E-01	1.1E-01	**3.6E-03**	**2.3E-03**	2.3E-03	2.2E-03	5.1E-03	4.7E-03
7.0	1.0E+00	1.0E+00	9.7E-01	3.7E-01	1.4E-02	**2.2E-03**	**2.1E-03**	**1.8E-03**	4.7E-03	3.0E-03
7.5	1.0E+00	1.0E+00	9.9E-01	6.5E-01	7.5E-02	3.2E-03	**1.7E-03**	**1.7E-03**	**1.7E-03**	3.3E-03
8.0	1.0E+00	1.0E+00	1.0E+00	8.2E-01	3.1E-01	1.4E-02	**1.9E-03**	**1.6E-03**	**1.5E-03**	**1.4E-03**
9.0	1.0E+00	1.0E+00	1.0E+00	9.9E-01	6.2E-01	1.5E-01	1.1E-02	**1.9E-03**	**1.6E-03**	**1.4E-03**
10.0	1.0E+00	1.0E+00	1.0E+00	1.0E+00	8.5E-01	3.9E-01	7.9E-02	9.0E-03	2.2E-03	**1.5E-03**
12.0	1.0E+00	1.0E+00	1.0E+00	1.0E+00	9.5E-01	7.3E-01	3.7E-01	1.0E-01	2.7E-02	5.7E-03
14.0	1.0E+00	1.0E+00	1.0E+00	1.0E+00	9.9E-01	8.7E-01	6.2E-01	2.4E-01	7.5E-02	2.3E-02
16.0	1.0E+00	1.0E+00	1.0E+00	1.0E+00	9.9E-01	9.2E-01	7.2E-01	3.5E-01	1.3E-01	5.7E-02
20.0	1.0E+00	1.0E+00	1.0E+00	1.0E+00	1.0E+00	9.7E-01	8.2E-01	4.9E-01	2.0E-01	9.2E-02

These 3D views give better insight about the right selection of parameters. For *nbit*, as shown in Figure 3.8, using higher values of *nbit* will always improve the performance. On the other hand, choosing the right value of *M* seems to be more difficult. For different values of *nbit*, there is a range of values of *M* that will result in optimal performance. For example, for *nbit*=5, *M* should range from 4 to 5.5 to get optimal performance. For *nbit*=9, *M* should be in the range of 6 to 8.5 in order to get the optimal performance.

3.5 Conclusion

In this chapter, the performance of semi-random LDPC codes has been simulated over AWGN in both floating-point representation and fixed-point representation. Different fixed-point parameters have been investigated. The code used was a 1/2-rate with block length of 1024. When selecting the maximum number of iterations It_{max}, it was shown that the performance will improve as we increase It_{max}. However, after reaching a value of about 60, little improvement will be gained for increasing It_{max} any further. From hardware perspective, reducing It_{max} will result in less decoding delay and more throughput. When selecting the number of precision bits *nbit*, performance will always improve with higher values of *nbit*. Nevertheless, the higher *nbit* is, the more costly hardware is going to be. Finally, selecting the value of the dynamic range *M* proved to be trickier. There is no fixed range or trend for selecting the optimal value of *M*. Getting the right value of *M* should be done in conjunction with *nbit*. For higher values of *nbit*, higher ranges of *M* will be chosen to get optimal performance.

CHAPTER 4

LDPC CODES: DESIGN AND PERFORMANCE

4.1 Introduction & Motivation

In most of the LDPC codes in the literature, the main target is usually to achieve the best possible performance in software simulations. In many cases, they end up with code designs that perform very close to Shannon limit but are almost impossible to realize in hardware. Since most error correcting codes are expected, eventually, to be used in practice and get implemented in hardware, this ultimate goal should affect the process of designing the code in the first place. The target in this thesis is to design LDPC codes with hardware in mind, i.e. design LDPC codes that have advantages for hardware implementation. Such advantage can come with cost of losing some performance. There are several approaches to achieve this goal. Some of them [31] start with a random **H** matrix and then apply certain procedures to reduce the maximum wire lengths. Other approaches build the **H** matrix in a structured way that limits the existence of long connections [32] [33].

4.2 Code Structure

The structure of the proposed LDPC code is shown in Figure 4.1.

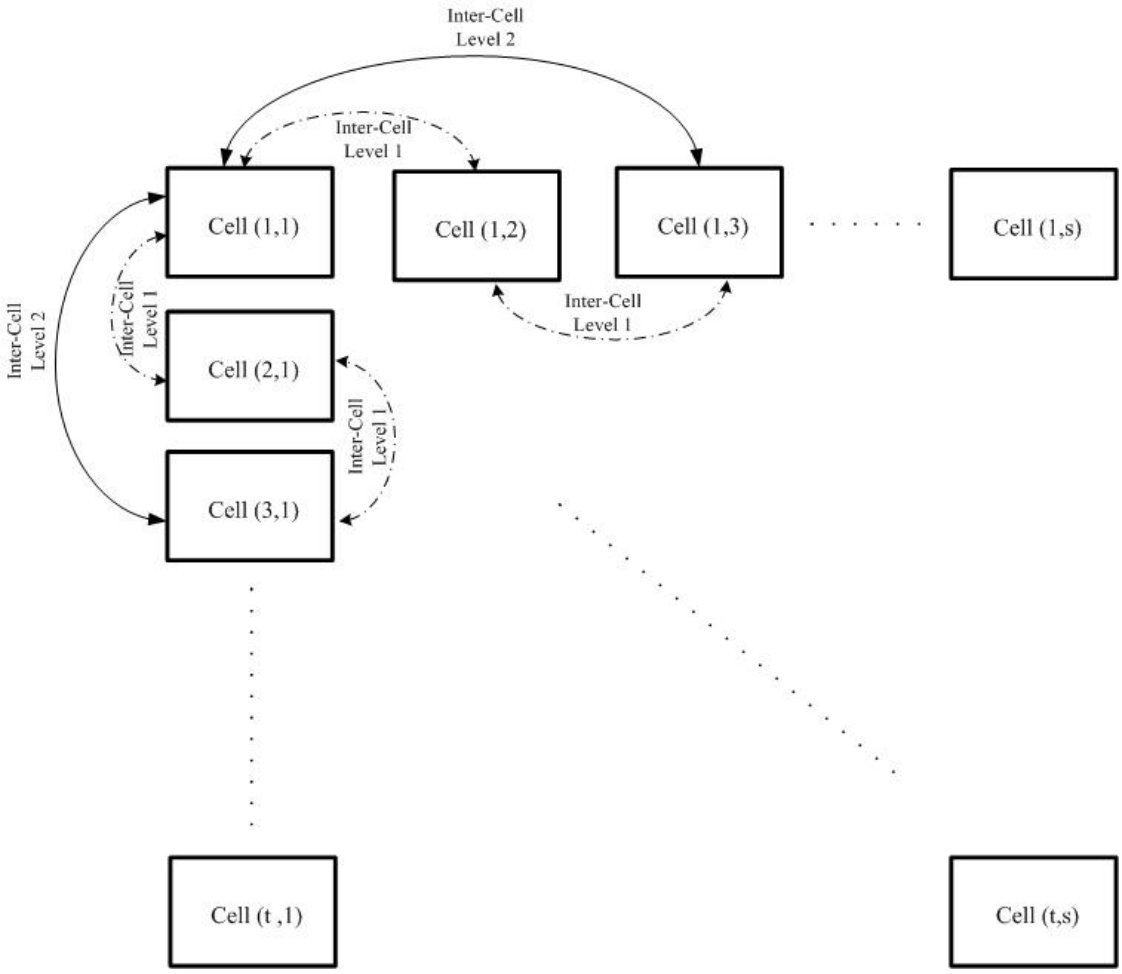

Figure 4.1 : Block diagram of the general structure of the code.

The code is formed by a lattice of cells. The lattice has *t* rows and *s* columns. Each cell contains a group of check nodes and variable nodes connected in a regular way. Examples of the structure of the cells are shown in Figure 4.2 and Figure 4.3.

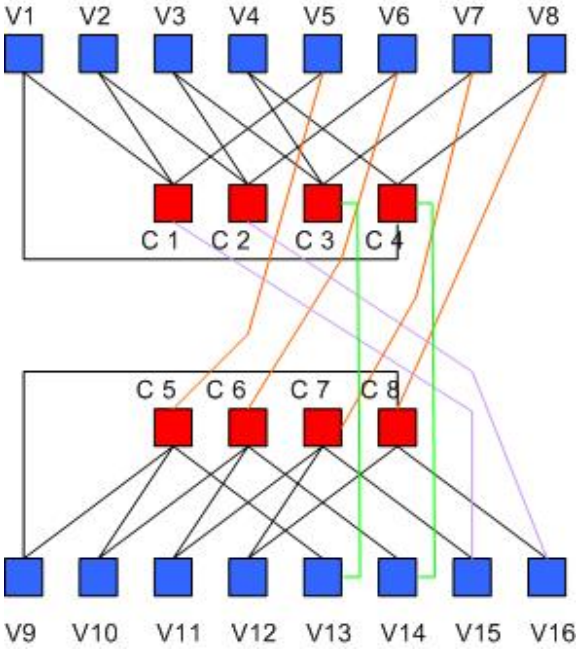

Figure 4.2 Cell with size 16x8

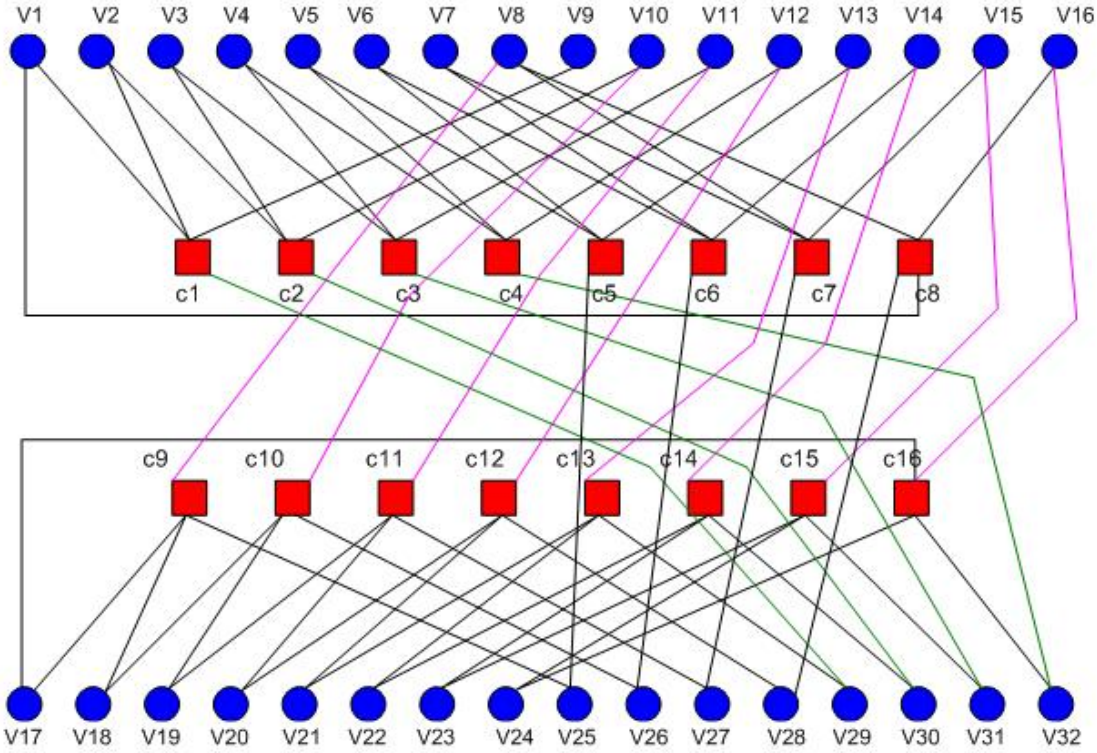

Figure 4.3 Cell with size 32x16

The code consists of a lattice of microcells. Each microcell consists of two groups of variable nodes and two groups of check nodes. The number of nodes in each group will vary depending on the code rate and the selection of the designer. Figure 4.2 shows a cell that has 8 check nodes and 16 variable nodes which can be used to construct a rate ½ code. Figure 4.3 shows another sample of cells which has 16 check nodes and 32 variable nodes which can be also used to construct a rate ½ code. Each microcell is connected to the adjacent microcells in structured way. The connections, also called edges, of this design can be classified into five types:

1. Intra-cell connections: which connect check nodes and variable nodes that exist in the same cell.
2. Horizontal inter-cell connections (level 1): which connect check and variable nodes that exist in adjacent cells in the same row of cells.
3. Vertical inter-cell connections (level 1): which connect check and variable nodes that exist in adjacent cells in the same column of cells.
4. Horizontal inter-cell connections (level 2): which connect check and variable nodes that exist in cells that are two columns apart in the same row of cells.
5. Vertical inter-cell connections (level 2): which connect check and variable nodes that exist in cells that are two rows apart in the same column of cells.

Figure 4.4 shows an example of this code. For clarity, the only inter-cell connections are only those connected to cell (1,1). This figure shows a factor graph of a (288,144). It consists of 9 cells arranges in a 3*3 lattice. Each cell has the size (32,16).

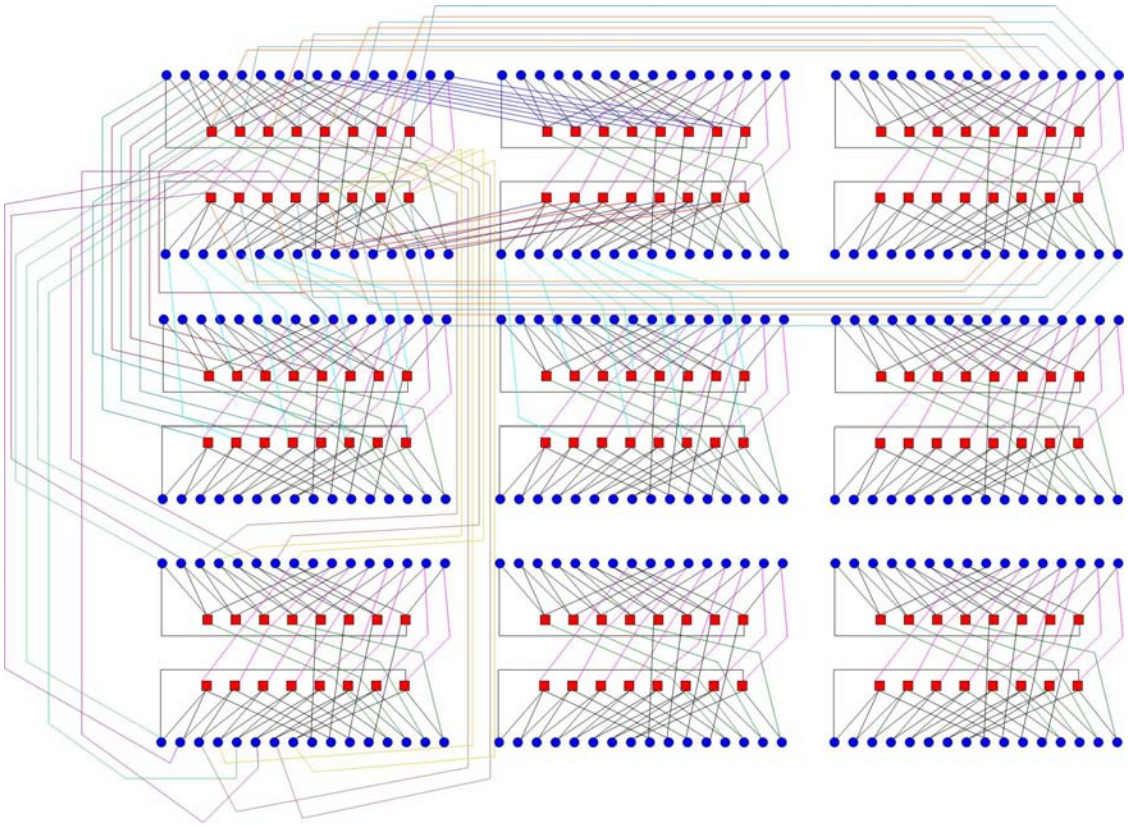

Figure 4.4 A 3x3 Lattice composed by 32x16 cells.

The lattice can be configured in many different arrangements for the same block length and code rate. For example, for a code with block length of 1024 and code rate ½, the lattice can take the shapes of 8x8 matrix, 16x4, 4x16 or 32x2 matrix when using the 16x8 cells. The code is going to be always irregular code. This design approach can also be used for different code rates. A very important feature of this design is that the complexity of wiring doesn't grow much when the size of the code grows up. This is because the inter-cell connections are limited to within 2 cells away. This feature will affect the hardware complexity of the LDPC decoder. This particular issue will be explored thoroughly in Section 5.3. Moreover, the maximum wire length is be limited to

the distance between two cells. This will greatly reduce the delay caused by the longest wire, which is expected with random code. This in turn will help improving the throughput of the system.

4.3 Assessment of the Complexity Reduction Advantage of the Proposed Code

4.3.1 Qualitative Assessment

Figure 4.5 and Figure 4.6 give an idea about the interconnect complexity of the proposed LDPC code in comparison with a random LDPC code. These figures show the interconnections after placement and before routing. It is obvious that the proposed LDPC code is less complex that the random LDPC code. Graphs of the interconnection complexity after routing are going to be shown in Chapter 5.

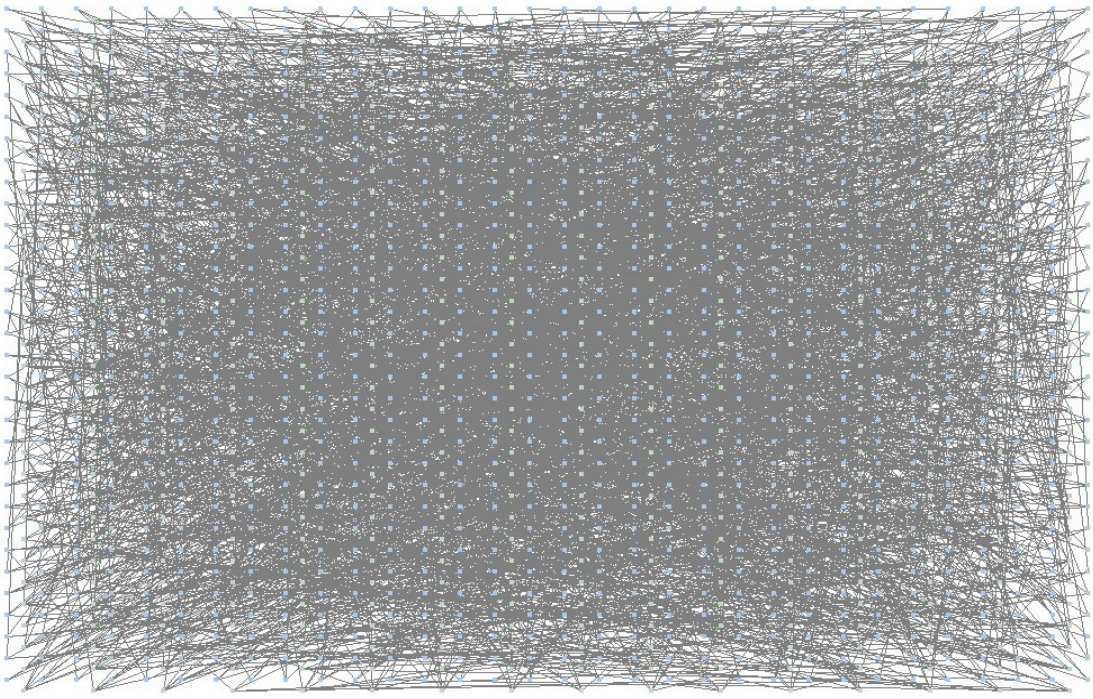

Figure 4.5 Interconnection complexity of a (512,1024) random LDPC code.

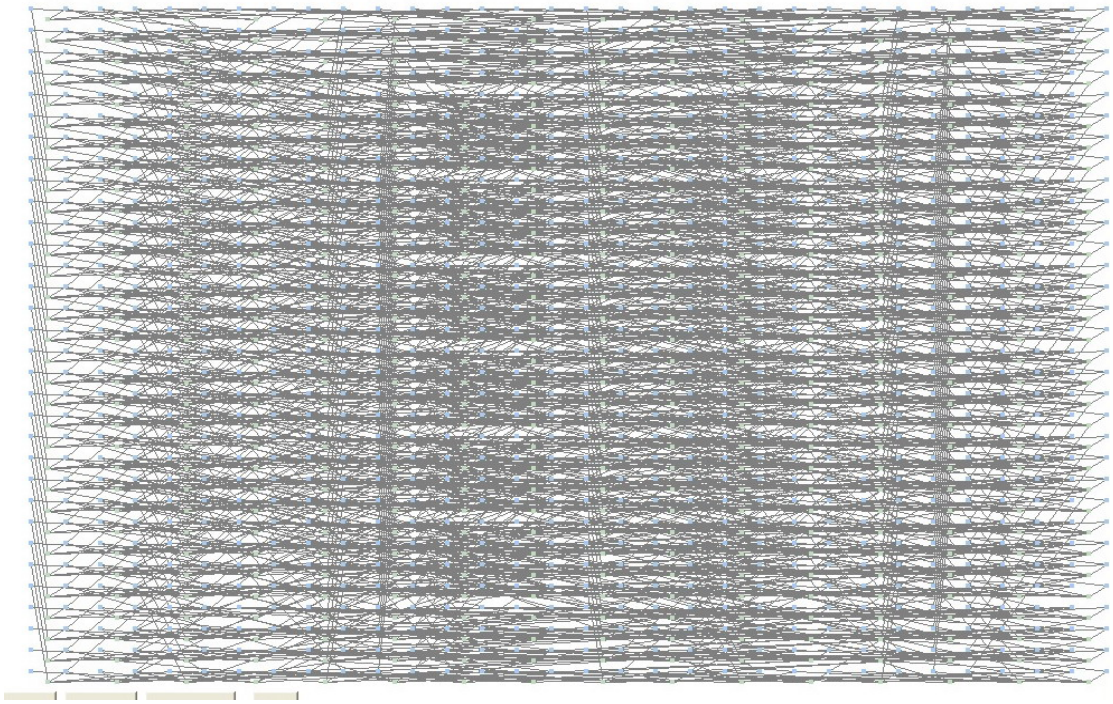

Figure 4.6 Interconnection complexity of a (512,1024) proposed LDPC code.

4.3.2 Quantitative Assessment

In order to quantitatively show the hardware advantage of the proposed LDPC code, a cost function will be used to quantify the complexity of the proposed LDPC code in comparison with another random LDPC code. The cost function will be the *Manhattan distance* of the placed graph of the specified LDPC code.

In order to calculate the *Manhattan distance*, the LDPC code should first be put into a placed graph form. The variable and check nodes are grouped in cells based on the code rate. For a $\frac{p}{q}$ rate LDPC code, each cell consists of p check nodes and q variable nodes. The cells are laid out on an X-Y grid area.

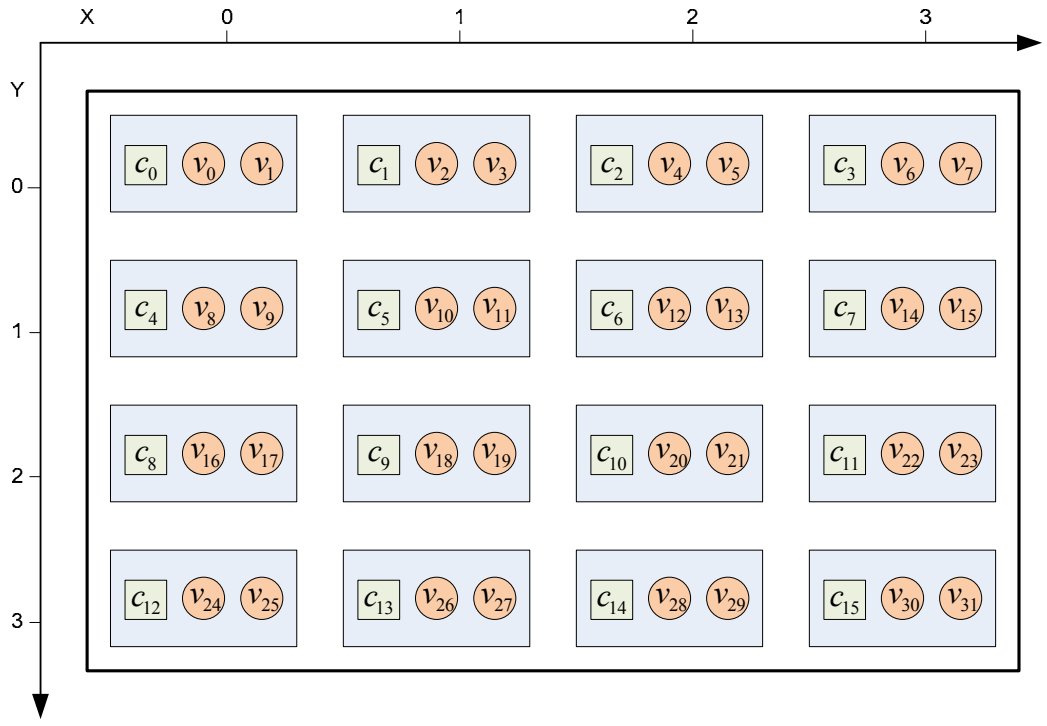

Figure 4.7 Example of a (32,16) LDPC code layout.

Figure 4.7 shows a layout example of a (32,16) LDPC code. This code has a rate of $\frac{1}{2}$, therefore, each cell contains one check node and two variable nodes. The cells are laid on a rectangular X-Y grid of size 4×4.

The *width* of the grid in which the cells are laid out is an input parameter to the proposed algorithm. The width of layout in the example of Figure 4.7 Example of a (32,16) LDPC code layout. is 4, this means that the x-coordinate of a cell can take one of the values from the set $\{0,1,2,3\}$. Based on the layout width, each node (variable or check) is assigned an (x,y) coordinate. The assignment of (x,y) coordinates of nodes of rate $\frac{p}{q}$ LDPC code is done as follows:

For a check node c_j, the (x,y) coordinates are given by:

$$c_j^x = \left\lfloor \frac{j}{p} \right\rfloor \bmod width \qquad (4.1)$$

$$c_j^y = \left\lfloor \frac{j}{p\times width} \right\rfloor \qquad (4.2)$$

Similarly, the (x,y) coordinates of a variable node v_i are given by:

$$v_i^x = \left\lfloor \frac{i}{q} \right\rfloor \bmod width \qquad (4.3)$$

$$v_i^y = \left\lfloor \frac{i}{q\times width} \right\rfloor \qquad (4.4)$$

For the layout example in Figure 4.7, the (x,y) coordinates of a check node c_j and a variable node v_i are: $c_j^x = j \bmod 4$, $c_j^y = \left\lfloor \frac{j}{4} \right\rfloor$, $v_i^x = \left\lfloor \frac{i}{2} \right\rfloor \bmod 4$ and $v_i^y = \left\lfloor \frac{i}{8} \right\rfloor$

The connection length between a variable node and a check node is computed based on the *Manhattan distance* between the two cells in which the nodes exist. The *Manhattan distance* (α) between a variable node v_i and a check node c_j is given by:

$$\alpha(v_i, c_j) = \left|v_i^x - c_j^x\right| + \left|v_i^y - c_j^y\right| \qquad (4.5)$$

For example, in Figure 4.7, the *Manhattan distance* between v_{10} and c_{11} is 3. The two nodes v_1 and c_0 have the same coordinates, therefore, the *Manhattan distance* between them is zero. Actually v_1 and c_0 reside in the same cell.

Figure 4.8 and Figure 4.9 show the results of Manhattan distance computations for different sizes of the proposed LDPC code (with w_r=6 and w_c=3) and a random LDPC codes (also with w_r=6 and w_c=3). The first graph shows that the average Manhattan distance of the proposed LDPC code almost stops increasing after size (1024,2048). On the other hand, the average Manhattan distance of the random LDPC code keeps increasing at a high pace. The same argument applies for the graph of the maximum Manhattan distance, shown in Figure 4.9. In this graph, we notice that the maximum Manhattan distance of the proposed LDPC code reaches its maximum at block length=512 and never increases beyond that. However, the maximum Manhattan distance of the random LDPC code keeps increasing at a high rate.

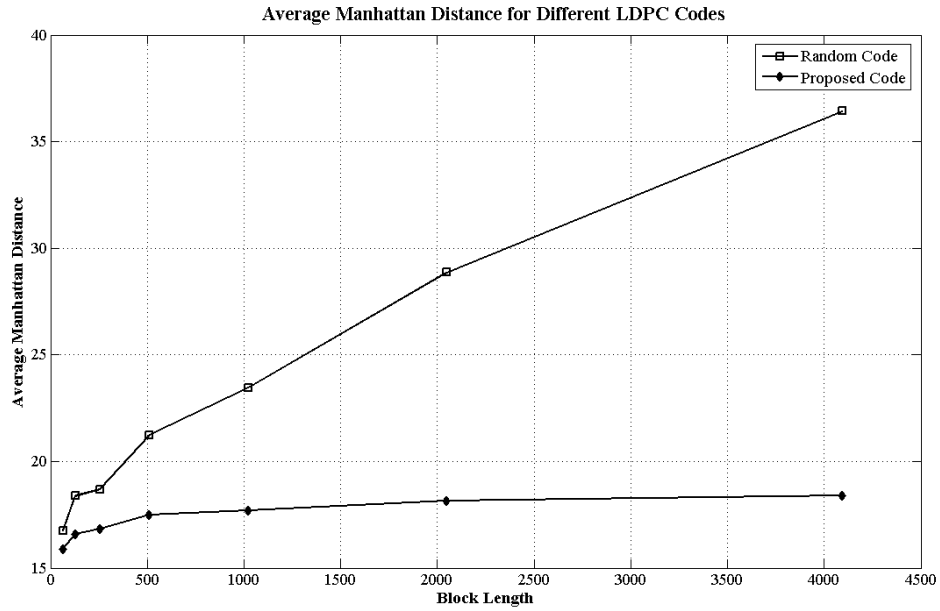

Figure 4.8 Average Manhattan distance for different LDPC codes.

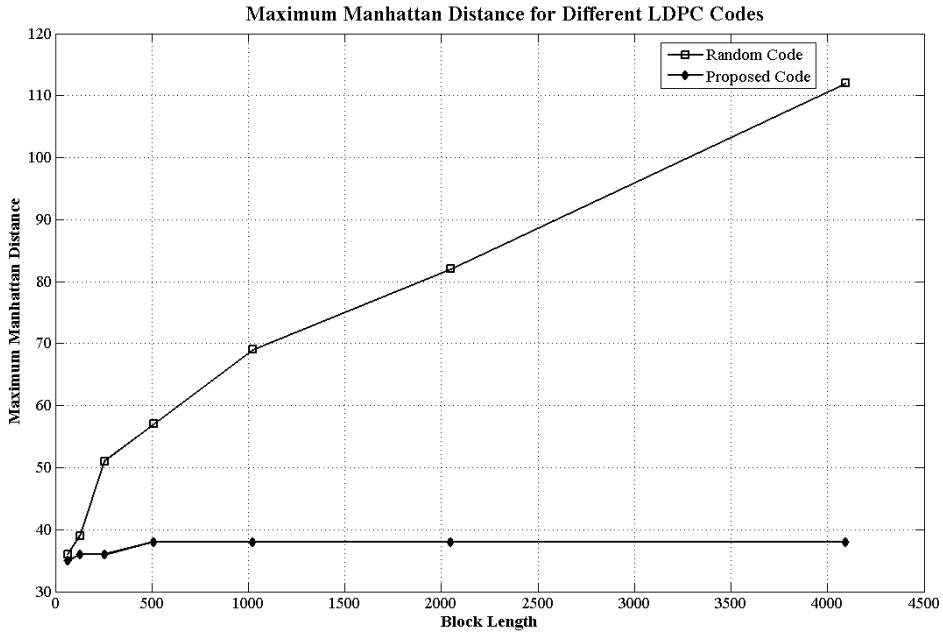

Figure 4.9 Maximum Manhattan distance for different LDPC codes.

4.4 Performance over AWGN.

The performance of the proposed LDPC code has been simulated for block length 1024, code rate ½, and minimum girth 8 and with an 8x8 lattice structure. The MATLAB file used to generate this code is listed in Appendix B. The FER performance is shown in Figure 4.10 and compared to other codes of the same block length.

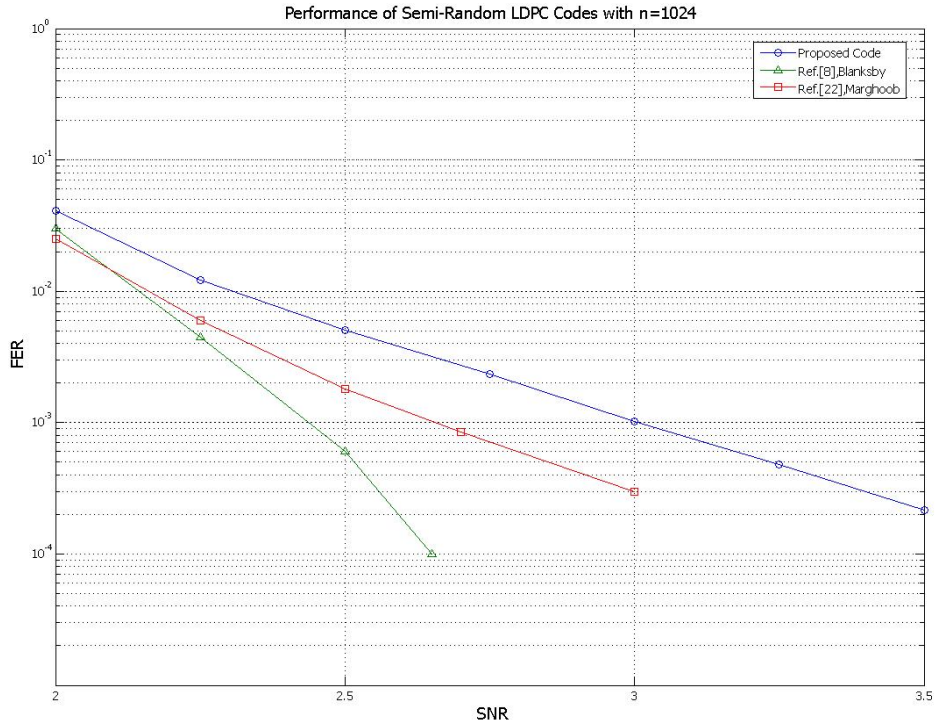

Figure 4.10 Performance of the proposed code compared to two similar codes.

Figure 4.10 shows the performance of the proposed code compared with the performance of two other codes with the same size [9] and [32]. In this plot, performance in measured in Frame Error Rate (FER) instead of Bit Error Rate (BER) in order to be able to compare our results with the references mentioned above. When FER=10^{-2}, the performance of the proposed code is less than the other two by 0.15dB. When FER=10^{-3}, the performance degrades more and the difference grows to 0.35dB and 0.6dB from the other codes. This

performance degradation is most likely caused by the trapping sets that are associated with the inherent locality of the code's structure. This problem can be reduced by using modified decoding algorithms as in [34]. Another way to mitigate this degradation is by carefully designing the cells and interchanging cell connections or by using larger cells.

4.5 Conclusion

In this chapter, an interconnect-efficient LDPC code has been proposed. The main goal of the design is to reduce the hardware complexity of the decoder. The proposed code is scalable and its complexity grows in a much less rate than random codes. The structure of the code has been explained with an example. The complexity reduction advantage of the proposed code has been demonstrated in two different ways. Graphs of post-placement designs shows that the wiring density of the proposed code is clearly much less than that of a random code. Moreover, quantitative analysis using *Manhattan distance* metric showed that the proposed code's average and maximum Manhattan distance saturates to a certain level while the values for the random code keeps increasing with the increase of the block length. A sample code with size (1024,512) have been simulated and compared with other codes and it showed some degradation in performance compared to the others. Some recent research [35] suggests that the reason behind that is the large number of trapping sets caused by the inherent locality of connection in the proposed code. However, the hardware complexity reduction, inherent in this code, is going to be explored thoroughly in Section 5.3.

CHAPTER 5

DESIGN AND IMPLEMENTATION OF THE LDPC DECODER

5.1 Introduction

In the chapter, the target is to design a hardware architecture for LDPC decoding and to implement it on a suitable platform. The decoder architecture can be split into three main parts:

- The check node unit.
- The variable node unit.
- The interconnecting network and the control logic.

In Section 5.2, the architecture of each of these parts will be designed depending on certain requirements. In Section 5.3, these architectures are going to be implemented in VHDL. In Section 5.4, the whole decoder is going to be synthesized on a hardware platform for two different LDPC codes.

5.2 Decoder Hardware Architecture

5.2.1 The Check Node Unit (CNU)

The check node unit should satisfy the following equation:

$$LLR^{(k)}(r_{ji}) = (-1)^{|L(j)|}\left(\prod_{i' \in L(j)\setminus\{i\}} sgn\left(LLR^{(k)}\left(q_{ji'}\right)\right)\right).j\left(\sum_{i' \in L(j)\setminus\{i\}} j\left(\left|LLR^{(k)}\left(q_{ji'}\right)\right|\right)\right) \quad (5.1)$$

where $j(x) = -\log(\tanh(\frac{x}{2}))$

So, the requirements of the design of the CNU can be specified as the following:

- Multiply the signs of the inputs.
- Apply the $\phi(x)$ on the magnitudes of the inputs.
- Add the outputs of $\phi(x)$ of the previous step.
- Avoid any overflow that might happen in the additions.
- After all additions finish, the resulting values must go through $\phi(x)$ and then to the output.

The proposed design of the CNU that satisfies these requirements is shown in Figure 5.1 The design shown here is a CNU3, which is a 3-input CNU.

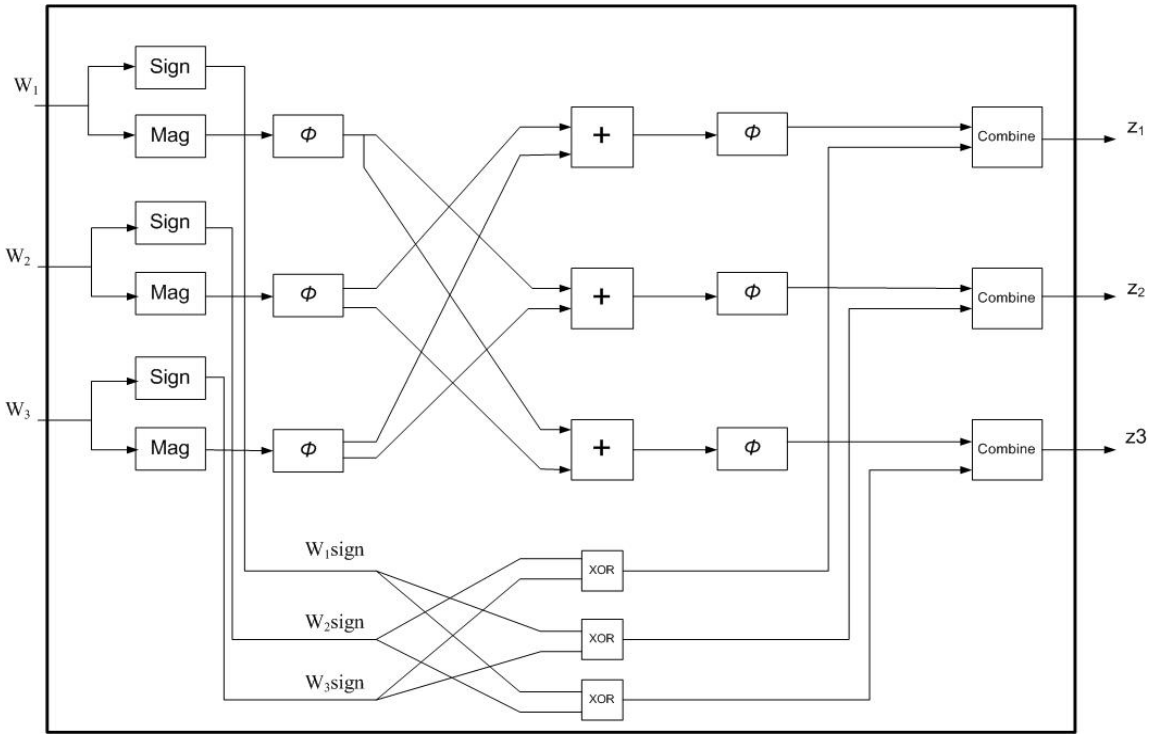

Figure 5.1 Check Node Unit (CNU) with 3 inputs.

This design is for a 3-input check node. The operation of this circuit can be described as follows:

- The input signals (w_1, w2 & w_3), which are 6-bit signals, are separated into two parts, sign and magnitude.
- The sign of each signal is gathered by taking the most significant bit (MSB) and then is fed to the XOR mesh.
- The remaining 5-bit magnitudes are fed into the $\phi(x)$ blocks.
- The $\phi(x)$ blocks are implemented using look-up tables (LUT's). Each LUT takes inputs of 5-bit resolution and, thus, contains 32 entries.
- After that, 2-input additions are done according to the equation. Overflow check is applied after every addition.
- After all additions finish, signals go again through $\phi(x)$ process.

- Signs of the inputs, taken in the first step, are taken into a mesh of XORs, and then combined with the outputs of the $\phi(x)$ to give the final outputs of the CNU3.

The same design is used for CNU4, CNU5, etc.

5.2.2 The Variable Node Unit (VNU)

The variable node unit (VNU) should satisfy the equation:

$$\mathrm{LLR}^{(k)}(q_{ji}) = \sum_{j' \in M(i) \setminus \{j\}} \mathrm{LLR}^{(k-1)}(r_{j'i}) + \mathrm{LLR}(p_i) \tag{5.2}$$

So, the requirements of the design of the VNU can be specified as the following:

- It basically contains addition processes.
- The values added can be either positive or negative
- Only two numbers should be added each time.

The proposed design of the VNU that satisfies these requirements is shown in Figure 5.2

The design shown here is a VNU2, which is a 2-input VNU.

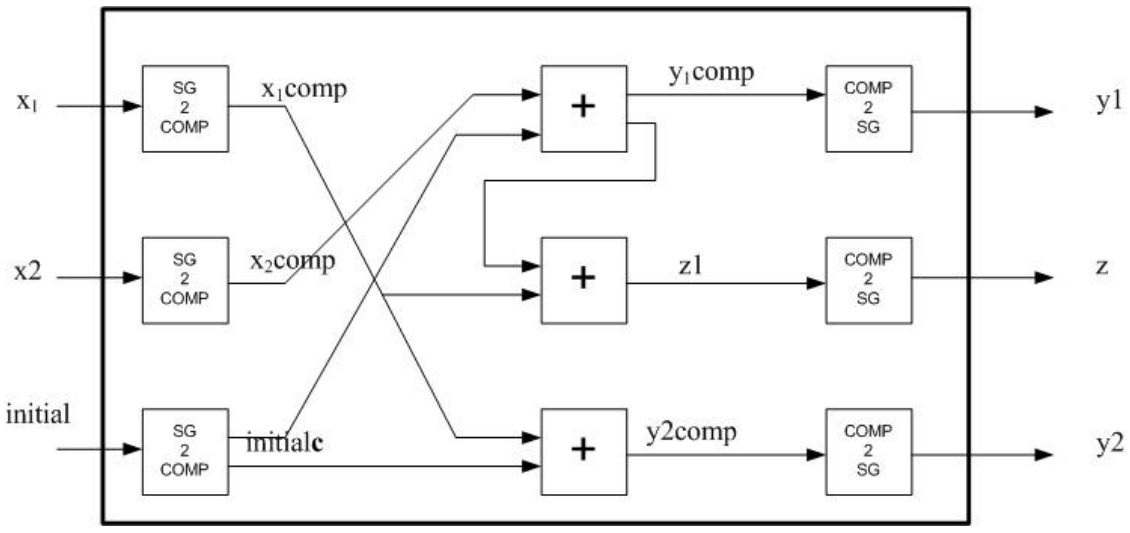

Figure 5.2 The Variable Node Unit (VNU) with 3 inputs.

The operation of this circuit can be described as follows:

- Addition can be done directly on the inputs, which can be either negative or positive. However, this will require having addition and subtraction operations which complicates the circuit.
- Doing the addition operation with the 2's complement eliminates the need for any subtraction blocks.
- The inputs are converted from sign-magnitude to 2's complement in the first stage.
- After that, addition operations are performed, two inputs at a time.
- Overflow check is done after each addition operation.
- After finishing all additions, the signals are converted from 2's complement format to sign-magnitude again.
- The message update is incorporated in the VNU because it has almost the same equation.

5.2.3 The Combined Decoder

The main unit of this LDPC decoder is required to do the following tasks:

1. Connecting the VNU's and the CNU's through the interconnecting network (defined by the **H** matrix).
2. Receiving the data from the previous stage and pipelining them to the VNU's at the right time.
3. Updating the values of the posterior information according to the equation:

$$LLR^{(k)}(q_i) = \sum_{j' \in M(i)} LLR^{(k)}(r_{j'i}) + LLR(p_i) \tag{5.3}$$

4. Performing the decoding process for a predefined number of iterations.

Figure 5.3 shows the decoder with all of its components.

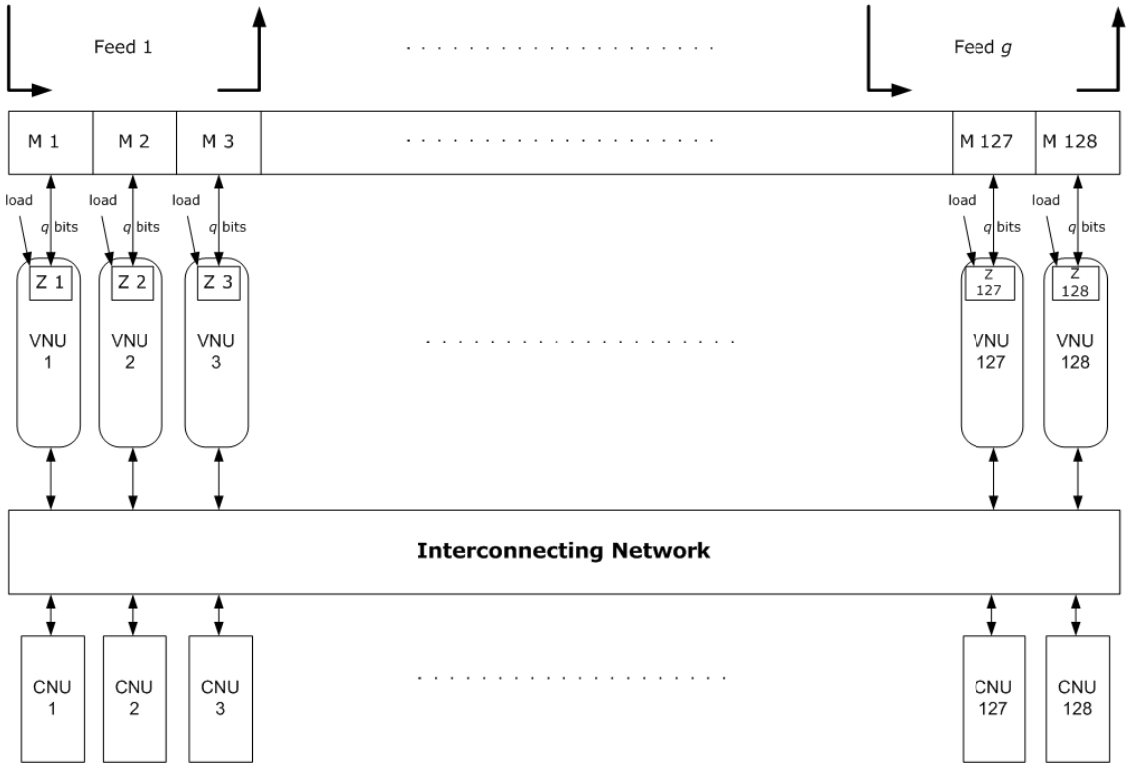

Figure 5.3 The Combined Decoder Architecture.

The decoder works in the following way:

1. Data messages are fed through different feed lines into registers (M1, M2, …., M128). This parallel pipelining is intended to speed up the data loading process.
2. When all messages are in the right position, the load signal is triggered and the messages are transferred to the Z registers.
3. The VNU's and the CNU's start the decoding process. Each iteration will take place in one clock cycle.
4. While the decoding process is ongoing, the next codeword is loaded into the *M* registers.

5. The values of the posterior information (Z's) are updated in every iteration.

After a specific number of iterations and when the next codeword is located precisely in the *M* registers, the data is exchanged between the Z registers and the *M* registers. This means that there will be no time wasted in feeding data into the variable nodes. Once a codeword is decoded completely, the next codeword is ready to start the decoding process.

In the above design, we can notice the following:

The number of iterations =

resolution of every message bit × the number of messages from every feed line

$$= (q \text{ bits/message}) \times (\text{messages/Feed line})$$

For example, if $q = 6$ and the required number of iterations is 24, then every feed line will carry 4 messages. This means that we will need 32 feed lines for this 128-bit decoder.

The number of iterations should be always an integer multiple of the message resolution.

The load signal is triggered by a counter adjusted according to the required number of iterations. After the last iteration, data is exchanged and the counter is set to 0 again.

The more data feeds we have, the more throughput we will get. This is going to be associated with less number of iterations. However, when the number of iterations goes higher than 60, the effect on performance will be negligible.

The interconnecting network is determined by the **H** matrix used. The structure of the **H** matrix has a huge effect on the complexity of the overall design.

5.3 VHDL Modeling and Verification

The LDPC decoder hardware structure described in the previous section has been implemented in VHDL. The VHDL code lists are shown in Appendix A. This VHDL

model is generic and can be used with any LDPC code. The general structure of the model is illustrated in Figure 5.4.

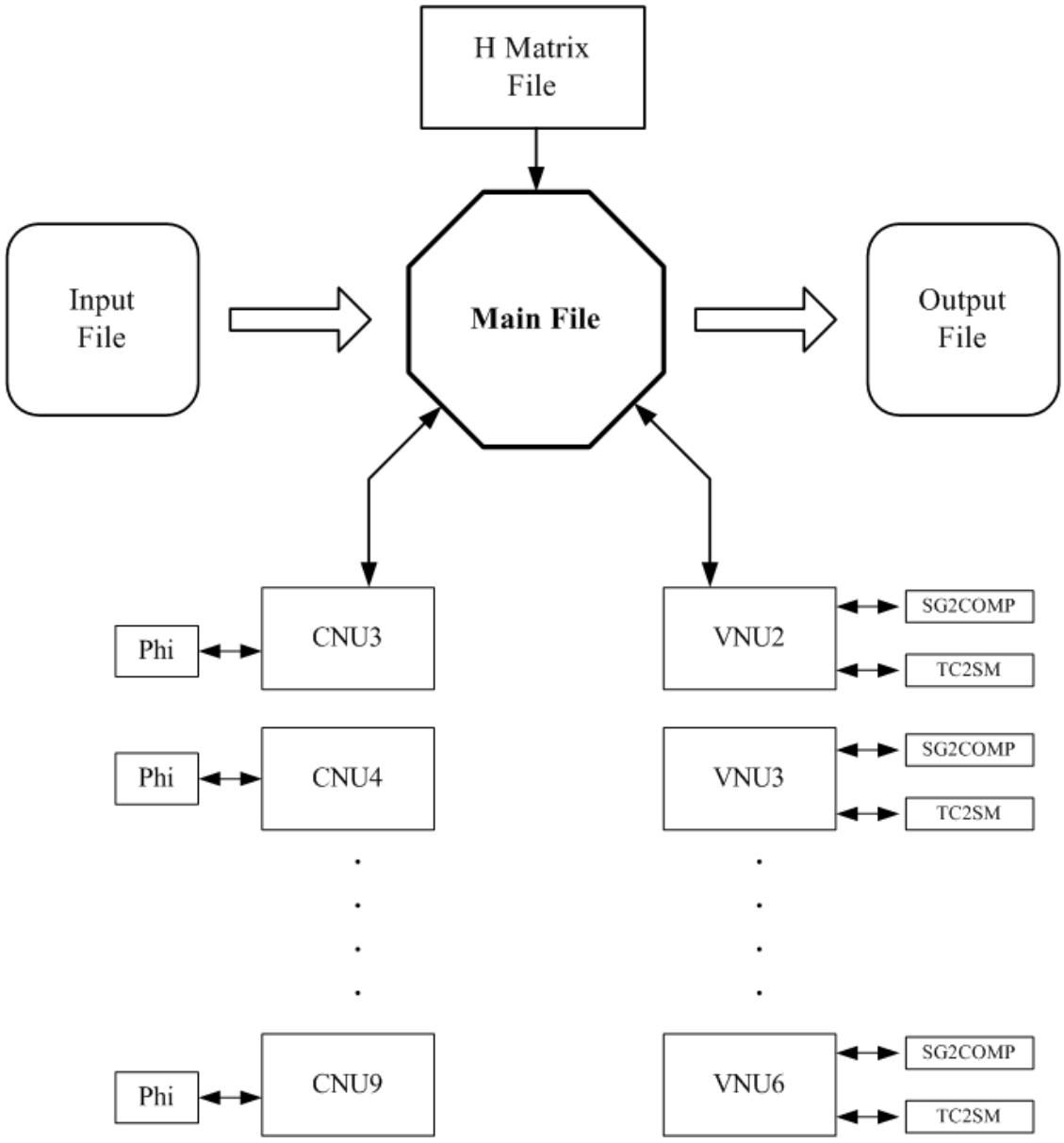

Figure 5.4 Structure of The VHDL model.

This VHDL model implements an LDPC decoder in a fully-parallel fashion. This of course will consume more hardware resources, compared to serial or serial-parallel approaches, but it will provide the maximum possible throughput. Since this model is

generic, it will deal with every **H** matrix regardless of any inherent regularity that the code might have (as in the code described in Chapter 4). However, implementation results will show some improvement for the proposed code's synthesis results compared to random ones.

The correctness of the VHDL code has been verified by comparing it with a fixed-point MATLAB code.

5.4 FPGA Synthesis of different LDPC codes

5.4.1 Synthesis Results

After designing the LDPC decoder and modeling it in VHDL, the next logical step is to synthesize it in hardware. The VHDL model describing the hardware architecture was synthesized over Xilinx VirtexE xcv3200e. The software used for the synthesis was Xilinx ISE 6. The purpose of synthesis is to get an idea about the different aspects of implementing our design on a hardware platform. Implementing one type of LDPC code might be enough, but implementing several LDPC codes with different code structures and different block lengths will give wider range of information. For this reason, four codes have been considered for synthesis. They represent two block lengths and two types of codes structures. The block lengths were 64 and 128. The LDPC code structures used in the synthesis were random LDPC codes and the proposed LDPC codes, which was described in Chapter 4. The synthesis results of the four codes are shown in TABLE 5.1

TABLE 5.1 Xilinx VirtexE Synthesis Results for Different LDPC Codes

		LDPC Code Structure			
		Proposed		Random	
Block Length		64	128	64	128
Equivalent Gate Count		115,440	282,807	196,944	N/A
No. of Slices		7,188	17,844	12,665	N/A
No. of LUTs	Total	13,813	34,280	24,460	N/A
	Logic	12,760	30,978	22,780	N/A
	Route-thru	1,053	3,302	1,680	N/A

For the proposed LDPC code, both block lengths where synthesized successfully. For the random LDPC code, only the shorter code, with block length of 64, has been synthesized. The random code with block length 128 was too big for this target platform. Snapshots of the post place-and-route circuit are shown in Figure 5.5 to Figure 5.10. These snapshots were taken using the Xilinx FPGA editor.

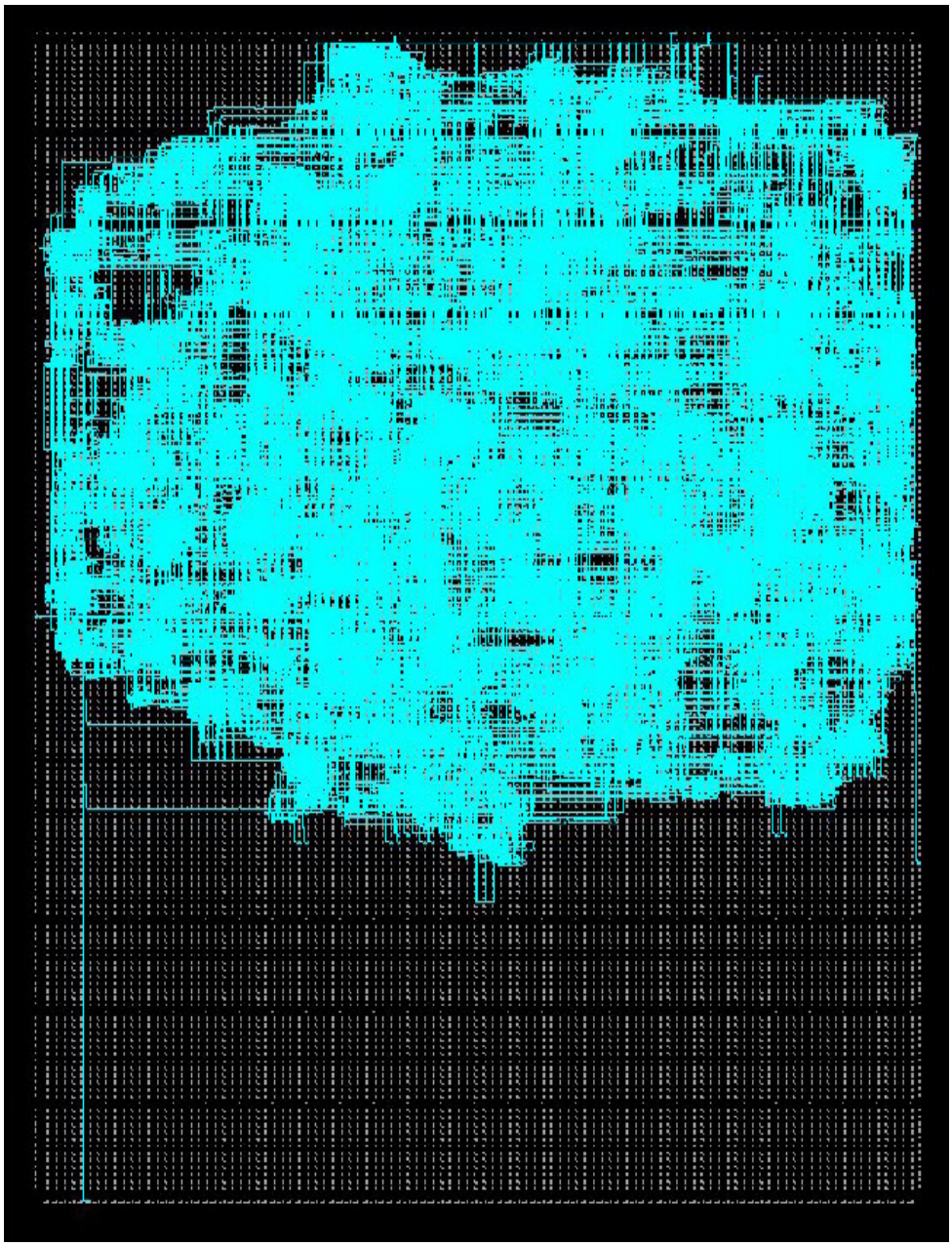

Figure 5.5 Overall view of synthesized proposed LDPC code with N=64

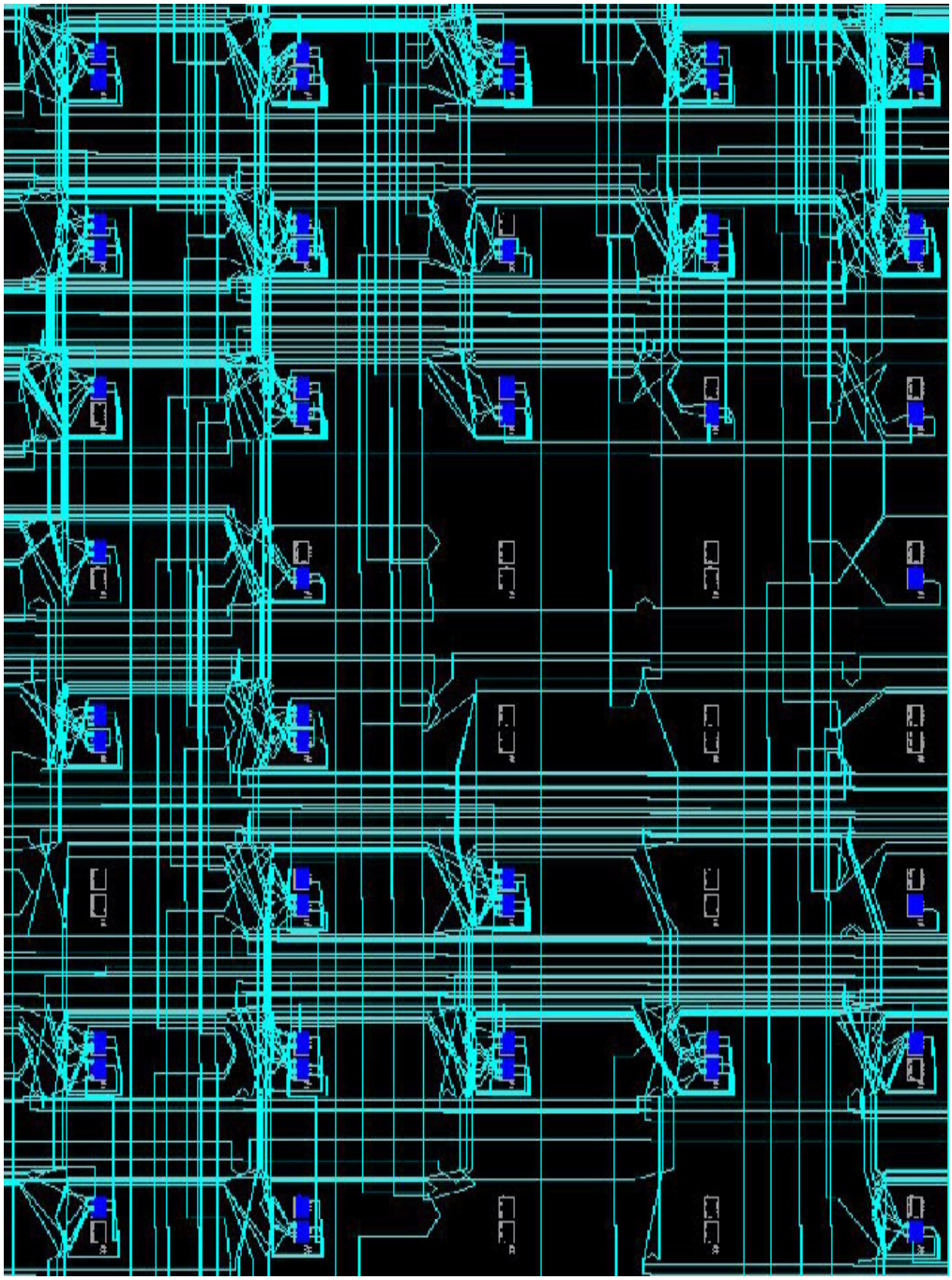

Figure 5.6 Closer view of the synthesized proposed LDPC code with N=64

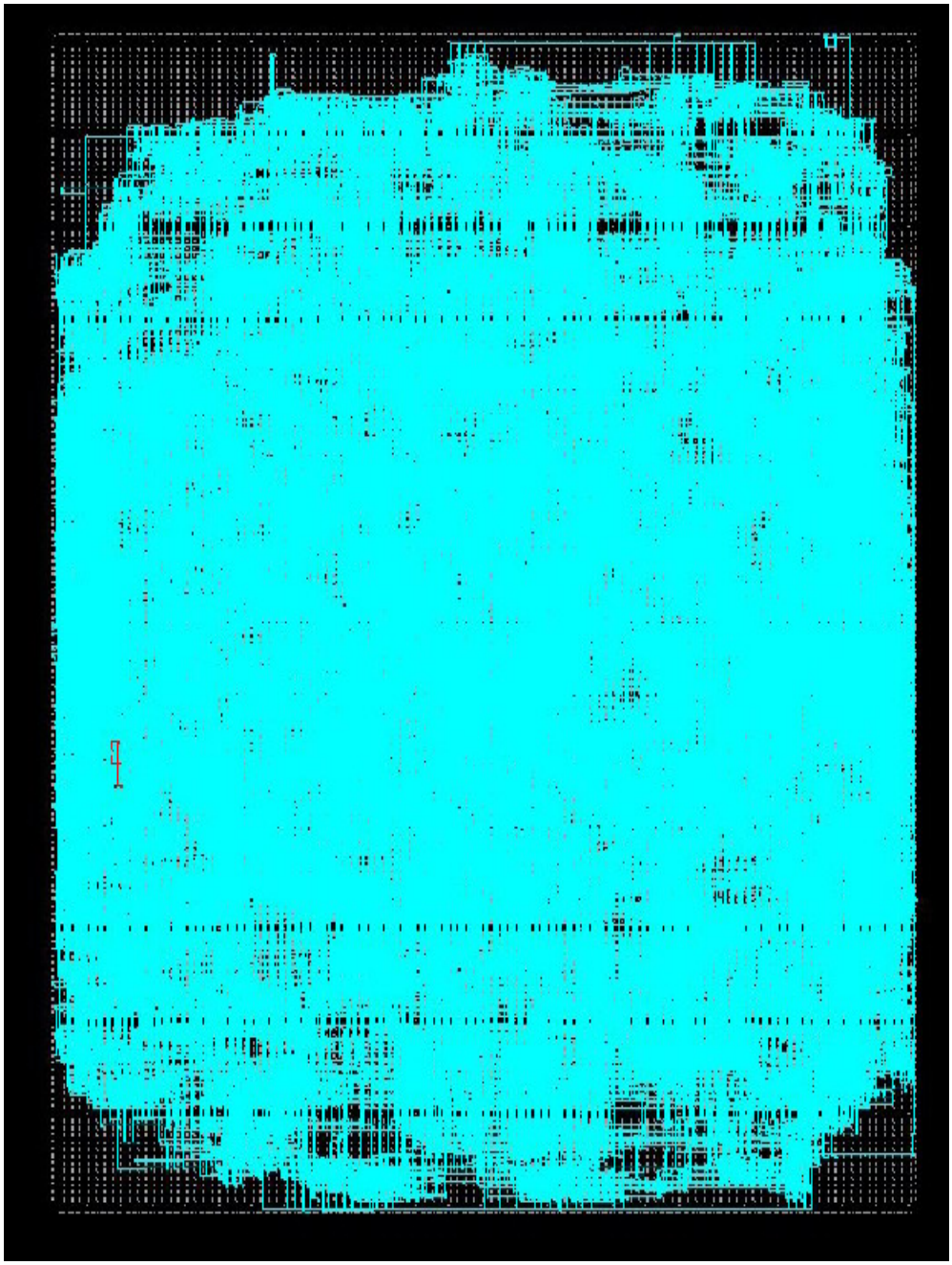

Figure 5.7 Overall view of the synthesized proposed LDPC code with N=128.

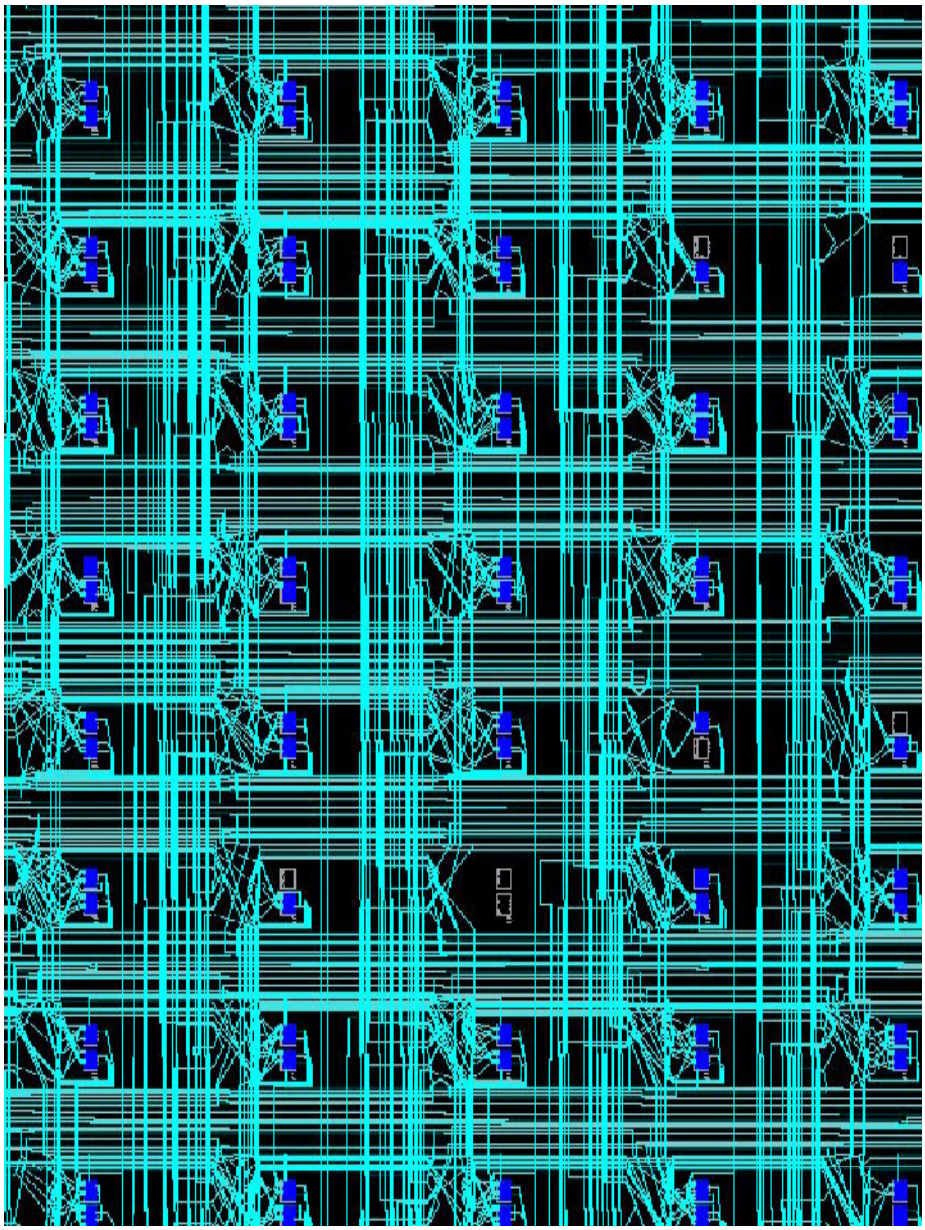

Figure 5.8 Closer view of the synthesized proposed LDPC code with N=128

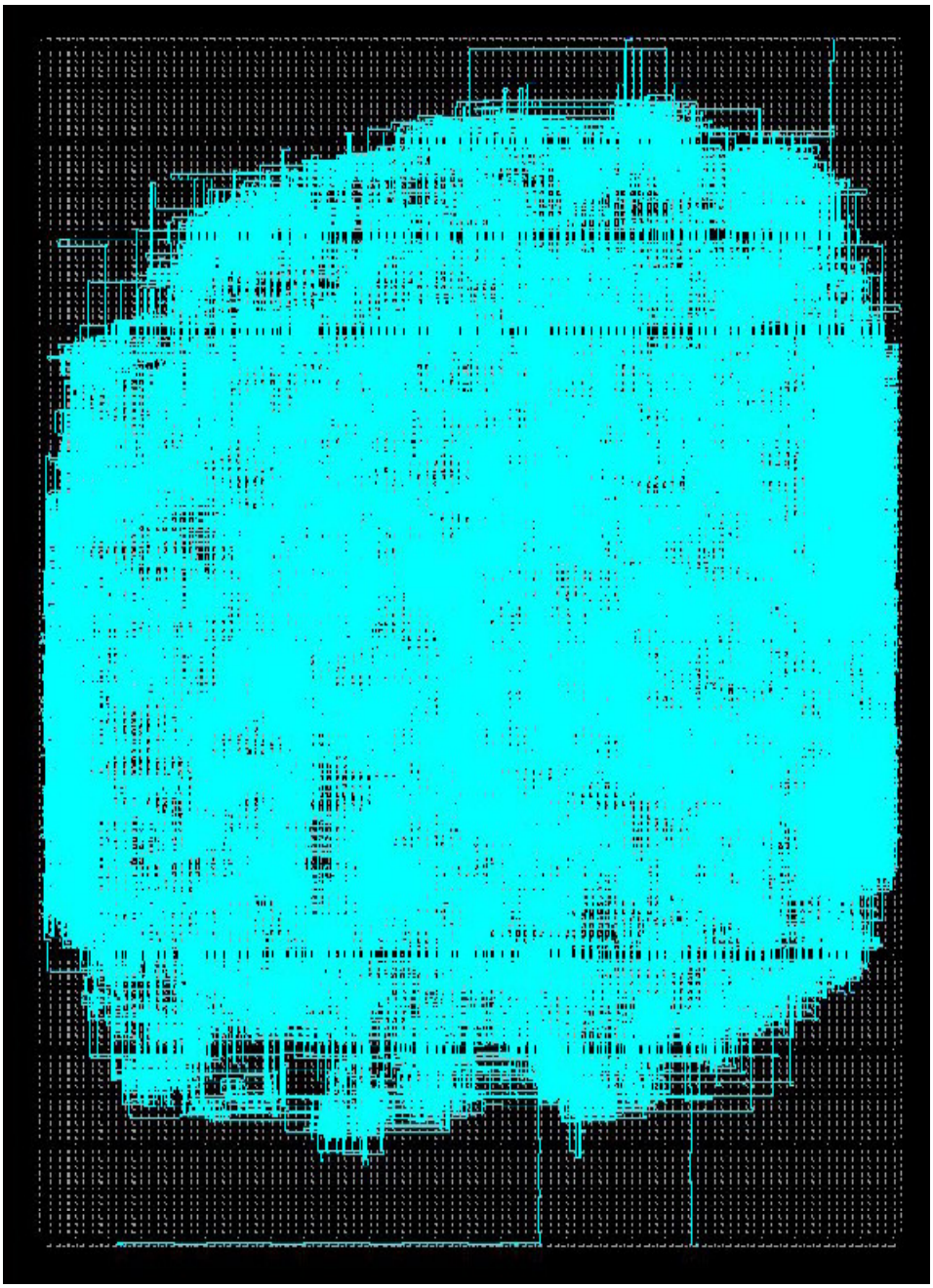

Figure 5.9 Overall view of the synthesized random LDPC code with N=64.

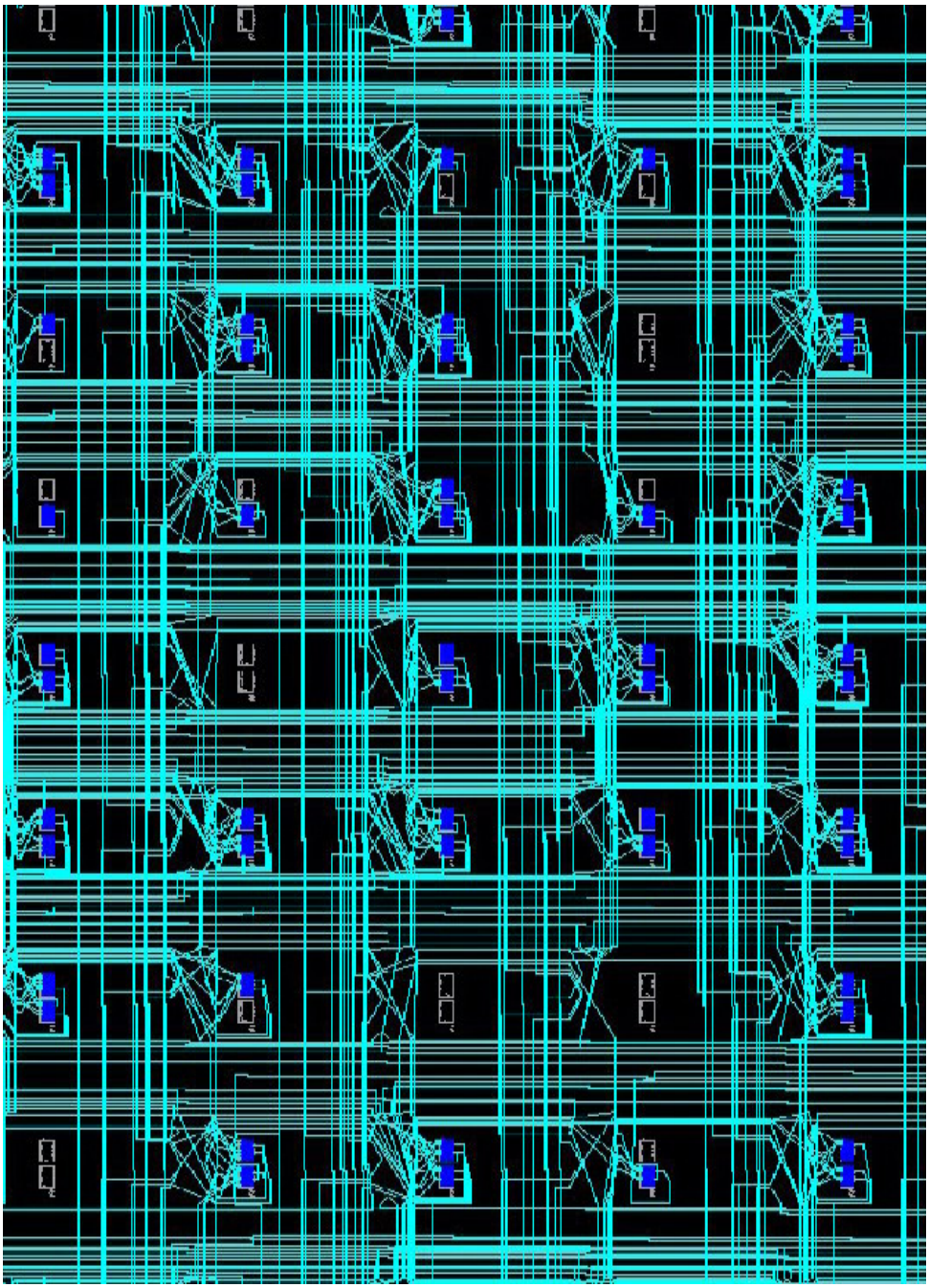

Figure 5.10 Closer view of the synthesized random LDPC code with N=64.

The analysis of these synthesis results can be done in two stages. The effect of the LDPC codes structure will be discussed first and next the effect of the block length will discussed.

5.4.2 Effect of the LDPC Code Structure on the Synthesis Results

From TABLE 5.1, we can extract the following table:

TABLE 5.2 Synthesis Results Comparison for n=64

		Code Structure		
		Proposed	Random	% Difference
Block Length		64	64	
Equivalent Gate Count		115,440	196,944	70.60
No. of Slices		7,188	12,665	76.20
No.of LUTs	Total	13,813	24,460	77.08
	Logic	12,760	22,780	78.53
	Route-thru	1,053	1,680	59.54

These two codes have the same block length (64), the same average check node degree (6) and the same average variable node degree (3). Thus, both codes have the same number of check nodes, variable nodes and edges. So, the implementation of the decoders representing these two codes will essentially require the same numbers of check node units, variable node units and the same number of wires connecting them. However, the only difference between them is the complexity of their wiring, which proved to have a big influence on the synthesis results. Looking at the table above (TABLE 5.2), we can deduce the following:

- The random code costs 76.2 % more slices than the proposed code.
- The random code costs 70.6 % more equivalent gate count than the proposed code.
- The random code costs 77.08% more LUT's than the proposed code.
- The random code costs 78.53% more logic LUT's than the proposed code.
- The random code costs 59.54% more route-thru LUT's than the proposed code.

The above numbers can be even clearer when comparing the figures of both codes. Thus, using the proposed code will save about 43% of the hardware resources compared with the random code, for this particular block length.

5.4.3 Effect of the LDPC Code Block length on the Synthesis Results:

From TABLE 5.1, the following table can be extracted:

TABLE 5.3 Synthesis Results of the Proposed code for different sizes

		Code Structure		
		Proposed	Proposed	% Difference
Block Length		64	128	
Equivalent Gate Count		115,440	282,807	144.98
No. of Slices		7,188	17,844	148.25
No. of LUTs	Total	13,813	34,280	148.17
	Logic	12,760	30,978	142.77
	Route-thru	1,053	3,302	213.58

These two codes have the same structure and they differ only in the block length. Doubling the block length will increase the number of slices by 148%, the number of gate count by 144% and the number of LUTs by 148%. The number of LUTs used as route-thru , that is used as wires rather that lookup tables, increased by 213%. This is because of the area limitation usually found in FPGAs. This area limitation problem reduces the utilization of the reduced interconnection complexity advantage inherent in the proposed LDPC code.

5.4.4 Throughput

In order to calculate the throughput of this design, we need to find the maximum delay, and then the maximum possible clock frequency. Then the throughput can be found using the relation:

$$Throughput = clock\ frequency \times bits\ processed\ every\ clock\ cycle \quad (5.4)$$

If we consider the 128-bit decoder illustrated in Figure 5.3, the throughput will depend on the number of iterations and the resolution of the message bits. The different selections of these two parameters will decide the number of feed lines and will result in different values of the throughput. This is shown in the following table. Note that the maximum delay of the decoder is 12.376 ns (taken from the Xilinx ISE 6.0 simulations), which means that the maximum clock frequency will be 80.8 MHz.

TABLE 5.4 Throughput of the 128-bit decoder for different parameters

No. of Iterations	Message Resolution q (bits)	Messages/Feed	No. of Feeds	Throughput (Mbit/s)
12	3	4	32	861.87
24	3	8	16	430.93
16	4	4	32	646.40
32	4	8	16	323.20
20	5	4	32	517.12
40	5	8	16	258.56
24	6	4	32	430.93
48	6	8	16	215.47

To be completely processed, each codeword requires an amount of time that equals the duration of all the iterations. If the number of iterations is 24, this means that the 128-bit codeword needs 24 clock cycles in order to be processed completely. So, 128 bits are processed in (24 × 12.376 ns). Each bit requires 2.35 ns which results in a throughput of 430.93 Mbit/s.

5.5 Conclusion

In this chapter, a generic LDPC decoder architecture has been designed and implemented. It can work with any LDPC code. This decoder has fully-parallel architecture in order to achieve maximum possible throughput. The decoder architecture consists of 3 parts: the check node unit (CNU), the variable node unit (VNU) and the main control and interconnecting unit. Each part has been designed, modeled in VHDL and synthesized on

an FPGA. The architecture has been implemented on FPGA for two types of codes: a random LDPC code and the proposed LDPC code discussed earlier in Chapter 4. The proposed code showed a significant complexity advantage compared to the random code. The hardware benefits of the proposed code could show more significant advantages if implemented using ASIC platform rather than FPGA. The throughput of this architecture, for the case of using the proposed code, was discussed for different configurations and different parameters and the post-routing interconnection complexity has been illustrated.

CHAPTER 6

SUMMARY OF RESULTS AND FUTURE WORK

In this thesis, several aspects related to hardware implementation of LDPC codes have been investigated. Fixed-point simulation has been used to investigate the appropriate selection of some hardware parameters before getting to the implementation stage. An interconnect-efficient code has been design with the intention to reduce the wiring complexity in the LDPC decoder. Finally, a fully-parallel LDPC decoder architecture has been modeled in VHDL and implemented in FPGA and tested some promised aspects of the proposed LDPC code.

In Chapter 3, the performance of semi-random LDPC codes has been simulated over AWGN in both floating-point representation and fixed-point representation. Different fixed-point parameters have been investigated. When selecting the maximum number of iterations It_{max}, it was shown that the performance will improve as we increase It_{max}. However, after reaching a value of about 60, little improvement will be gained for increasing It_{max} any further. From hardware perspective, reducing It_{max} will result in less decoding delay and more throughput. When selecting the number of precision bits *nbit*, performance has been shown to improve with higher values of *nbit*. Nevertheless, the higher *nbit* is, the more costly hardware is going to be. Finally, selecting the value of the dynamic range *M* proved to be more involved. Getting the right value of *M* should be done in conjunction with *nbit*. For higher values of *nbit*, higher ranges of *M* will be

chosen to get optimal performance. Fixed-point analysis can be very valuable for selecting some hardware parameters before getting into actual implementation. This will also give an idea about how much degradation can happen when changing any of the hardware parameters.

In Chapter 4, an interconnect-efficient LDPC code has been proposed. The main goal of the design was to reduce the hardware complexity of the LDPC decoder. The proposed code is scalable and its complexity grows in a much less rate than random codes. The structure of the code has been explained with an example. The complexity reduction advantage of the proposed code have been demonstrated qualitatively and quantitatively. Graphs of post-placement designs have shown that the wiring density of the proposed code is clearly much less than that of a random code. Moreover, quantitative analysis using *Manhattan distance* metric showed that the proposed code's average an maximum Manhattan distance saturates to a certain level while the values for the random code keeps increasing with the increase of the block length. A sample code with size (1024,512) have been simulated and compared with other codes and it showed some degradation in performance compared to the others. The performance degradation was resulted from the large number of trapping sets inherent in the proposed code.

In Chapter 5, a generic LDPC decoder architecture has been designed and implemented. The target was to get the maximum possible throughput, which led to the choice of fully-parallel architecture. The design can work with any LDPC code. The decoder architecture consists of 3 parts: the check node unit (CNU), the variable node unit (VNU) and the main control and interconnecting unit. Each part has been designed, modeled in VHDL

and synthesized on an FPGA. The architecture has been implemented on FPGA for two types of codes: a random LDPC code and the proposed LDPC code discussed earlier in Chapter 4. The proposed code showed a significant complexity advantage compared to the random code. The hardware benefits of the proposed code could show more significant advantages if implemented using ASIC platform rather than FPGA. The throughput of this architecture, for the case of using the proposed code, was discussed for different configurations and different parameters and the post-routing interconnection complexity has been illustrated.

The results of this thesis can lead to several suggestions for possible future work.

- The fixed-point simulation can be further investigated using different ways in selecting the hardware parameters. For example, nonlinear quantization levels can be used instead of linear.
- The proposed code showed some promising advantages but its performance was degraded by the excess amount of trapping sets. So, a possible future work may be to elaborate on the same design approach and find ways that can eliminate the effect of trapping sets while preserving most of the promising features.
- The LDPC decoder hardware architecture investigated and implemented in this thesis is a generic one. Designing specific architectures for certain codes, such as the one proposed in this thesis, can result in even more reduction in hardware complexity.

Appendix A
VHDL Code for The LDPC Decoder

A.1 The Overall VHDL Model

The VHDL model consists of several files that can vary depending on the size and type of the **H** matrix. The main file contains the interconnection logic and controls the input and output processes. The main file uses the **H** matrix file to construct the interconnecting network of the LDPC decoder. The **H** matrix file will specify how many CNU's and VNU's are going to be used, the degree of each one and the wiring between them. Figure A.1 shows the relation between the different files of the VHDL model. The CNU's shown in Figure A.1 go from CNU3 up to CNU9, but the can be easily expanded to any required degree. This also applies to the VNU's.

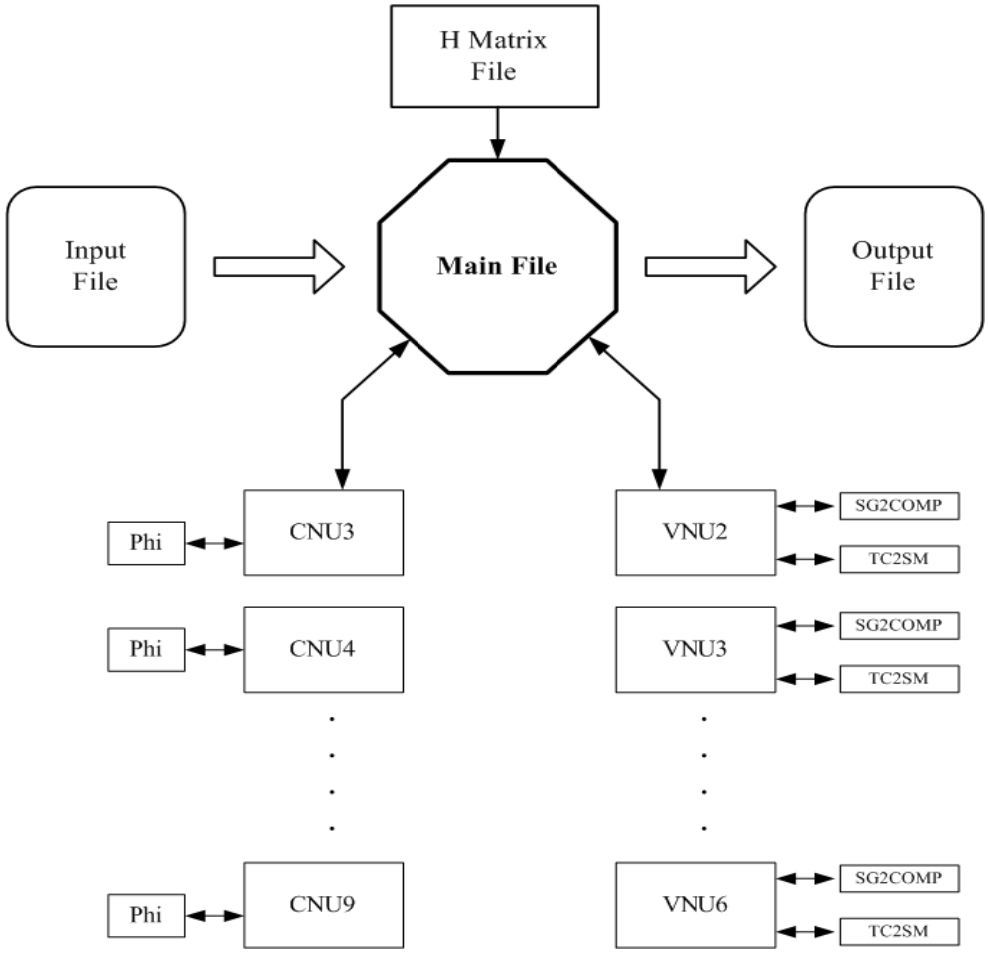

Figure A.1: Structure of The VHDL Model Files

A.2 The Main File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.hmatrix.all;

entity ldpc_con4 is
generic (n: natural :=6; noofmsg: natural :=6; noofchk :natural :=3;
chkdeg: natural :=3; msgdeg: natural :=2);
      Port (
            clk, load, sel, sin : in std_logic;
```

```
            sout : out std_logic
           );
end ldpc_con4;

architecture Behavioral of ldpc_con4 is


subtype ms1 is std_logic_vector(0 to n-1);
type ms2 is array (0 to msgdeg-1) of ms1;
type ms3 is array (0 to noofmsg-1) of ms2;
signal min, mout : ms3;

subtype r1 is std_logic_vector(0 to n-1);
type r2 is array (0 to noofmsg-1) of r1;
signal rout, z: r2;

subtype cs1 is std_logic_vector(0 to n-1);
type cs2 is array (0 to chkdeg-1) of cs1;
type cs3 is array (0 to noofchk-1) of cs2;
signal cin, cout : cs3;

signal t : std_logic_vector(0 to noofmsg);

component Check_Node9
generic ( n : natural :=6);
PORT (w1,w2,w3,w4,w5,w6,w7,w8,w9 :in  std_logic_vector(n-1 downto 0);
     z1,z2,z3,z4,z5,z6,z7,z8,z9 : out std_logic_vector(n-1 downto 0));
END component ;

component Check_Node8
generic ( n : natural :=6);
PORT (w1,w2,w3,w4,w5,w6,w7,w8 :in  std_logic_vector(n-1 downto 0);
     z1,z2,z3,z4,z5,z6,z7,z8 : out std_logic_vector(n-1 downto 0));
END component ;

component Check_Node7
generic ( n : natural :=6);
PORT (w1,w2,w3,w4,w5,w6,w7 :in  std_logic_vector(n-1 downto 0);
     z1,z2,z3,z4,z5,z6,z7 : out std_logic_vector(n-1 downto 0));
END component ;

component Check_Node6
generic ( n : natural :=6);
PORT (w1,w2,w3,w4,w5,w6 :in  std_logic_vector(n-1 downto 0);
     z1,z2,z3,z4,z5,z6 : out std_logic_vector(n-1 downto 0));
END component ;


component Check_Node5
generic ( n : natural :=6);
PORT (w1,w2,w3,w4,w5 :in  std_logic_vector(n-1 downto 0);
     z1,z2,z3,z4,z5 : out std_logic_vector(n-1 downto 0));
END component ;

component Check_Node4
```

```
generic ( n : natural :=6);
PORT (w1,w2,w3,w4 :in  std_logic_vector(n-1 downto 0);
     z1,z2,z3,z4 : out std_logic_vector(n-1 downto 0));
END component ;

component Check_Node3
generic ( n : natural :=6);
PORT (w1,w2,w3 :in  std_logic_vector(n-1 downto 0);
     z1,z2,z3 : out std_logic_vector(n-1 downto 0));
END component ;

component Variable_Node5
generic ( n : natural :=6);
PORT (clk,load:in std_logic; initial,x1,x2,x3,x4,x5 :in
std_logic_vector(n-1 downto 0);
     y1,y2,y3,y4,y5, z : out std_logic_vector(n-1 downto 0));
END component ;

component Variable_Node4
generic ( n : natural :=6);
PORT (clk,load:in std_logic; initial,x1,x2,x3,x4 :in
std_logic_vector(n-1 downto 0);
     y1,y2,y3,y4, z : out std_logic_vector(n-1 downto 0));
END component ;

component Variable_Node3
generic ( n : natural :=6);
PORT (clk,load:in std_logic; initial,x1,x2,x3 :in std_logic_vector(n-1
downto 0);
     y1,y2,y3, z : out std_logic_vector(n-1 downto 0));
END component ;

component Variable_Node2
generic ( n : natural :=6);
PORT (clk,load:in std_logic; initial,x1,x2 :in std_logic_vector(n-1
downto 0);
     y1,y2, z: out std_logic_vector(n-1 downto 0));
END component ;

component Variable_Node1
generic ( n : natural :=6);
PORT (clk,load:in std_logic; initial,x1 :in std_logic_vector(n-1 downto
0);
     y1, z: out std_logic_vector(n-1 downto 0));
END component ;


component SREG
  Generic (n: natural := 6);
  port (clk, sel, sin : in std_logic; d: in std_logic_vector(n-1 downto
0);  q : buffer std_logic_vector(n-1 downto 0) );
end component;


begin
```

```
t(0) <= sin;
MSGnodes: for m1 in 0 to noofmsg-1 generate
      tm5: if mdegree(m1) = 5 Generate
              reg5: SREG generic map(n)  port map(clk, sel, t(m1),
z(m1) , rout(m1) );
              t(m1+1) <= rout(m1)(n-1);
            nodesmsg5: Variable_Node5  port map(clk, load, rout(m1),
min(m1)(0),min(m1)(1),min(m1)(2), min(m1)(3),
min(m1)(4),mout(m1)(0),mout(m1)(1),mout(m1)(2),
mout(m1)(3),mout(m1)(4), z(m1));
      end generate;
      tm4: if mdegree(m1) = 4 Generate
              reg4: SREG generic map(n)  port map(clk, sel, t(m1),
z(m1) , rout(m1) );
              t(m1+1) <= rout(m1)(n-1);
            nodesmsg4: Variable_Node4  port map(clk, load, rout(m1),
min(m1)(0),min(m1)(1),min(m1)(2), min(m1)(3),
mout(m1)(0),mout(m1)(1),mout(m1)(2), mout(m1)(3), z(m1));
      end generate;
      tm3: if mdegree(m1) = 3 Generate
              reg3: SREG generic map(n)  port map(clk, sel, t(m1),
z(m1) , rout(m1) );
              t(m1+1) <= rout(m1)(n-1);
            nodesmsg3: Variable_Node3  port map(clk, load, rout(m1),
min(m1)(0),min(m1)(1),min(m1)(2),mout(m1)(0),mout(m1)(1),mout(m1)(2),
z(m1));
      end generate;
      tm2: if mdegree(m1) = 2 Generate
             reg2: SREG generic map(n)  port map(clk, sel, t(m1), z(m1)
, rout(m1) );
              t(m1+1) <= rout(m1)(n-1);
            nodesmsg2: Variable_Node2  port map(clk, load, rout(m1),
min(m1)(0),min(m1)(1),mout(m1)(0),mout(m1)(1), z(m1));
      end generate;
      tm1: if mdegree(m1) = 1 Generate
              reg1: SREG generic map(n)  port map(clk, sel, t(m1), z(m1)
, rout(m1) );
              t(m1+1) <= rout(m1)(n-1);
            nodesmsg1: Variable_Node1  port map(clk, load, rout(m1),
min(m1)(0), mout(m1)(0), z(m1));
      end generate;
end generate;

--sout <= rout(noofmsg-1)(mdegree(noofmsg-1)-1);
sout <= rout(noofmsg-1)(n-1);


CHKnodes: for n1 in 0 to noofchk-1 generate
      tc9: if cdegree(n1) = 9 Generate
            nodeschk9: Check_Node9 port
map(cin(n1)(0),cin(n1)(1),cin(n1)(2),cin(n1)(3),cin(n1)(4),cin(n1)(5),c
in(n1)(6),cin(n1)(7),cin(n1)(8),cout(n1)(0),cout(n1)(1),cout(n1)(2),cou
t(n1)(3),cout(n1)(4),cout(n1)(5),cout(n1)(6),cout(n1)(7),cout(n1)(8));
      end generate;
      tc8: if cdegree(n1) = 8 Generate
```

```
            nodeschk8: Check_Node8 port
map(cin(n1)(0),cin(n1)(1),cin(n1)(2),cin(n1)(3),cin(n1)(4),cin(n1)(5),c
in(n1)(6),cin(n1)(7),cout(n1)(0),cout(n1)(1),cout(n1)(2),cout(n1)(3),co
ut(n1)(4),cout(n1)(5),cout(n1)(6),cout(n1)(7));
      end generate;
      tc7: if cdegree(n1) = 7 Generate
            nodeschk7: Check_Node7 port
map(cin(n1)(0),cin(n1)(1),cin(n1)(2),cin(n1)(3),cin(n1)(4),cin(n1)(5),c
in(n1)(6),cout(n1)(0),cout(n1)(1),cout(n1)(2),cout(n1)(3),cout(n1)(4),c
out(n1)(5),cout(n1)(6));
      end generate;
      tc6: if cdegree(n1) = 6 Generate
            nodeschk6: Check_Node6 port
map(cin(n1)(0),cin(n1)(1),cin(n1)(2),cin(n1)(3),cin(n1)(4),cin(n1)(5),c
out(n1)(0),cout(n1)(1),cout(n1)(2),cout(n1)(3),cout(n1)(4),cout(n1)(5))
;
      end generate;
      tc5: if cdegree(n1) = 5 Generate
            nodesch5: Check_Node5 port
map(cin(n1)(0),cin(n1)(1),cin(n1)(2),cin(n1)(3),cin(n1)(4),cout(n1)(0),
cout(n1)(1),cout(n1)(2),cout(n1)(3),cout(n1)(4));
      end generate;
      tc4: if cdegree(n1) = 4 Generate
            nodesch4: Check_Node4 port
map(cin(n1)(0),cin(n1)(1),cin(n1)(2),cin(n1)(3),cout(n1)(0),cout(n1)(1)
,cout(n1)(2),cout(n1)(3));
      end generate;
      tc3: if cdegree(n1) = 3 Generate
            nodesch3: Check_Node3 port
map(cin(n1)(0),cin(n1)(1),cin(n1)(2),cout(n1)(0),cout(n1)(1),cout(n1)(2
));
      end generate;
end generate;

--mapping code
--main algorithm

process(clk, mout, cout)

variable  ccounter: ccount;
variable  mcounter: mcount;

begin

for j1 in 0 to (noofchk-1) loop
      ccounter(j1):=0;
end loop;

for i1 in 0 to (noofmsg-1) loop
      mcounter(i1) :=0;
end loop;

for j1 in 0 to (noofchk-1) loop
  for i1 in 0 to (noofmsg-1) loop
     if hsparse(j1)(i1)='1' then
         cin(j1)(ccounter(j1)) <= mout(i1)(mcounter(i1));
```

```
         min(i1)(mcounter(i1)) <= cout(j1)(ccounter(j1));
         ccounter(j1) := ccounter(j1)+1;
         mcounter(i1):= mcounter(i1)+1;
       end if;
end loop;
end loop;

end process;

end Behavioral;
```

A.3 The Matrix File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

Package hmatrix is
subtype hbasic is std_logic;
constant noofmsg: integer :=6;
constant noofchk: integer :=3;

type onerow is array (0 to noofmsg-1) of hbasic;
type compH is array (0 to noofchk-1) of onerow;

Constant hsparse: compH :=(
 ('1','1','0','1','0','0') ,
 ('0','1','1','0','1','0') ,
 ('1','0','1','0','0','1')
 );

type mcount is array (0 to noofmsg-1) of integer;
type ccount is array (0 to noofchk-1) of integer;

constant cdegree: ccount := ( 3, 3 , 3);
constant mdegree: mcount := ( 2, 2,2, 2, 2, 2);
end hmatrix;
```

Note that this file is for a small **H** matrix in order to avoid using a huge space.

A.4 The CNU Files

A.4.1 CNU3

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

ENTITY Check_Node3 IS
generic ( n : natural :=6);
PORT (w1,w2,w3 :in  std_logic_vector(n-1 downto 0);
     z1,z2,z3 : out std_logic_vector(n-1 downto 0));
END Check_Node3 ;

ARCHITECTURE CNU OF Check_Node3 IS
 signal w1mag: std_logic_vector(n-2 downto 0);
 signal w2mag: std_logic_vector(n-2 downto 0);
 signal w3mag: std_logic_vector(n-2 downto 0);

        -- Defining the phi function---
subtype bit5 is std_logic_vector(n-2 downto 0);
type  bit5_Array is array(0 to 2**(n-1)-1  ) of bit5;
subtype Rinteger is integer range 0 to 2**(n-1)-1;

Function To_Integer (Bin : bit5) Return RInteger IS
      Variable Result: RInteger;
Begin
      Result := 0;
      For I IN Bin'RANGE Loop
            If Bin(I) = '1' then
                  Result := Result + 2**I;
            End if;
      End Loop;
      Return Result;
End To_Integer;

function phi (a:bit5) return bit5 is

constant phi_table: bit5_Array :=(
     ("11111"),
     ("10110"),
     ("10001"),
     ("01101"),
     ("01011"),
     ("01010"),
     ("01000"),
     ("00111"),
     ("00110"),
     ("00101"),
     ("00101"),
     ("00100"),
     ("00100"),
     ("00011"),
     ("00011"),
     ("00010"),
     ("00010"),
     ("00010"),
     ("00010"),
     ("00001"),
     ("00001"),
```

```
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"));
begin
return phi_table( To_Integer(a) );
end phi;

signal w1magt1 : std_logic_vector(n-1 downto 0);
signal w1magt5 : std_logic_vector(n-2 downto 0);
signal w1magt6 : std_logic_vector(n-2 downto 0);

signal w2magt1 : std_logic_vector(n-1 downto 0);
signal w2magt5 : std_logic_vector(n-2 downto 0);
signal w2magt6 : std_logic_vector(n-2 downto 0);

signal w3magt1 : std_logic_vector(n-1 downto 0);
signal w3magt5 : std_logic_vector(n-2 downto 0);
signal w3magt6 : std_logic_vector(n-2 downto 0);

BEGIN

w1mag<=w1(n-2 downto 0);
w2mag<=w2(n-2 downto 0);
w3mag<=w3(n-2 downto 0);

  w1magt1<=('0' & phi(w2mag))+ ('0' & phi(w3mag));
  w1magt5<= "11111" when w1magt1(n-1) ='1' else w1magt1(n-2 downto 0);
  w1magt6<=phi(w1magt5) ;
  z1<=((w2(n-1) xor w3(n-1) xor '1' )& w1magt6(n-2 downto 0)) ;

  w2magt1<=('0' & phi(w1mag))+ ('0' & phi(w3mag));
  w2magt5<= "11111" when w2magt1(n-1)='1' else w2magt1(n-2 downto 0);
  w2magt6<=phi(w2magt5) ;
  z2<=((w1(n-1) xor w3(n-1) xor '1' )& w2magt6(n-2 downto 0));

  w3magt1<=('0' & phi(w1mag))+ ('0' & phi(w2mag));
  w3magt5<= "11111" when w3magt1(n-1)='1' else w3magt1(n-2 downto 0);
  w3magt6<=phi(w3magt5) ;
  z3<=((w1(n-1) xor w2(n-1) xor '1' )& w3magt6(n-2 downto 0)) ;

END CNU;
```

A.4.2 CNU4

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

ENTITY Check_Node4 IS
generic ( n : natural :=6);
PORT (w1,w2,w3,w4 :in  std_logic_vector(n-1 downto 0);
     z1,z2,z3,z4 : out std_logic_vector(n-1 downto 0));
END Check_Node4 ;

ARCHITECTURE CNU OF Check_Node4 IS
 signal w1mag: std_logic_vector(n-2 downto 0);
 signal w2mag: std_logic_vector(n-2 downto 0);
 signal w3mag: std_logic_vector(n-2 downto 0);
 signal w4mag: std_logic_vector(n-2 downto 0);

        -- Defining the phi function---
subtype bit5 is std_logic_vector(n-2 downto 0);
type  bit5_Array is array(0 to 2**(n-1)-1  ) of bit5;
subtype Rinteger is integer range 0 to 2**(n-1)-1;

Function To_Integer (Bin : bit5) Return RInteger IS
      Variable Result: RInteger;
Begin
      Result := 0;
      For I IN Bin'RANGE Loop
            If Bin(I) = '1' then
                  Result := Result + 2**I;
            End if;
      End Loop;
      Return Result;
End To_Integer;

function phi (a:bit5) return bit5 is

constant phi_table: bit5_Array :=(
     ("11111"),
     ("10110"),
     ("10001"),
     ("01101"),
     ("01011"),
     ("01010"),
     ("01000"),
     ("00111"),
     ("00110"),
     ("00101"),
     ("00101"),
     ("00100"),
     ("00100"),
     ("00011"),
     ("00011"),
     ("00010"),
     ("00010"),
     ("00010"),
     ("00010"),
     ("00001"),
```

```
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00001"),
     ("00000"),
     ("00000"),
     ("00000"),
     ("00000"),
     ("00000"));
begin
return phi_table( To_Integer(a) );
end phi;

signal w1magt1 : std_logic_vector(n-1 downto 0);
signal w1magt11 : std_logic_vector(n-2 downto 0);
signal w1magt2 : std_logic_vector(n-1 downto 0);
signal w1magt5 : std_logic_vector(n-2 downto 0);
signal w1magt6 : std_logic_vector(n-2 downto 0);

signal w2magt1 : std_logic_vector(n-1 downto 0);
signal w2magt11 : std_logic_vector(n-2 downto 0);
signal w2magt2 : std_logic_vector(n-1 downto 0);
signal w2magt5 : std_logic_vector(n-2 downto 0);
signal w2magt6 : std_logic_vector(n-2 downto 0);

signal w3magt1 : std_logic_vector(n-1 downto 0);
signal w3magt11 : std_logic_vector(n-2 downto 0);
signal w3magt2 : std_logic_vector(n-1 downto 0);
signal w3magt5 : std_logic_vector(n-2 downto 0);
signal w3magt6 : std_logic_vector(n-2 downto 0);

signal w4magt1 : std_logic_vector(n-1 downto 0);
signal w4magt11 : std_logic_vector(n-2 downto 0);
signal w4magt2 : std_logic_vector(n-1 downto 0);
signal w4magt5 : std_logic_vector(n-2 downto 0);
signal w4magt6 : std_logic_vector(n-2 downto 0);

BEGIN

w1mag<=w1(n-2 downto 0);
w2mag<=w2(n-2 downto 0);
w3mag<=w3(n-2 downto 0);
w4mag<=w4(n-2 downto 0);

  w1magt1<=('0' & phi(w2mag))+ ('0' & phi(w3mag));
  w1magt11<= "11111" when w1magt1(n-1)='1' else w1magt1(n-2 downto 0);
  w1magt2<=('0' & w1magt11(n-2 downto 0))+ ('0' & phi(w4mag));
  w1magt5<= "11111" when w1magt2(n-1)='1' else w1magt2(n-2 downto 0);
  w1magt6<=phi(w1magt5) ;
  z1<=((w2(n-1) xor w3(n-1) xor w4(n-1) )& w1magt6(n-2 downto 0)) ;

  w2magt1<=('0' & phi(w1mag))+ ('0' & phi(w3mag));
  w2magt11<= "11111" when w2magt1(n-1)='1' else w2magt1(n-2 downto 0);
```

```
  w2magt2<=('0' & w2magt11(n-2 downto 0))+ ('0' & phi(w4mag));
  w2magt5<= "11111" when w2magt2(n-1)='1' else w2magt2(n-2 downto 0);
  w2magt6<=phi(w2magt5) ;
  z2<=((w1(n-1) xor w3(n-1) xor w4(n-1) )& w2magt6(n-2 downto 0));

  w3magt1<=('0' & phi(w1mag))+ ('0' & phi(w2mag));
  w3magt11<= "11111" when w3magt1(n-1)='1' else w3magt1(n-2 downto 0);
  w3magt2<=('0' & w3magt11(n-2 downto 0))+ ('0' & phi(w4mag));
  w3magt5<= "11111" when w3magt2(n-1)='1' else w3magt2(n-2 downto 0);
  w3magt6<=phi(w3magt5) ;
  z3<=((w1(n-1) xor w2(n-1) xor w4(n-1) )& w3magt6(n-2 downto 0)) ;

  w4magt1<=('0' & phi(w1mag))+ ('0' & phi(w2mag));
  w4magt11<= "11111" when w4magt1(n-1)='1' else w4magt1(n-2 downto 0);
  w4magt2<=('0' & w4magt11(n-2 downto 0))+ ('0' & phi(w3mag));
  w4magt5<= "11111" when w4magt2(n-1)='1' else w4magt2(n-2 downto 0);
  w4magt6<=phi(w4magt5) ;
  z4<=((w1(n-1) xor w2(n-1) xor w3(n-1) )& w4magt6(n-2 downto 0)) ;

END CNU;
```

A.5 The VNU Files

A.5.1 VNU2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

ENTITY Variable_Node2 IS
generic ( n : natural :=6);
PORT (clk,load:in std_logic; initial,x1,x2 :in std_logic_vector(n-1
downto 0);
     y1,y2, z: out std_logic_vector(n-1 downto 0));
END Variable_Node2;

ARCHITECTURE VNU OF Variable_Node2 IS

component SG2COMP
generic ( n : natural :=6);
    port (x : in std_logic_vector(n-1 downto 0);
          y : out std_logic_vector(n-1 downto 0));
end component;
 signal x1comp,x2comp,y1comp,y2comp,ytemp1,ytemp2, y1compt, y2compt :
std_logic_vector(n-1 downto 0);
 signal initvalue, initialc, z1, z2 : std_logic_vector(n-1 downto 0);
 signal ovf1, ovf2, ovf3 : std_logic;

BEGIN
gi: SG2COMP port map (initial, initialc);

init: process(clk)
begin
if (load='1' and clk'event and clk='1' ) then
```

```
   initvalue<= initialc;
end if;
end process;

u1: SG2COMP port map(x1, x1comp);
u2: SG2COMP port map(x2, x2comp);

   y1comp<= x2comp+initvalue;
   y2comp<= x1comp+initvalue;
   z1 <= y1comp + x1comp;

   ovf1 <= ( not (x2comp(n-1) xor initvalue(n-1))) and  (y1comp(n-1)
xor x2comp(n-1)) ;
   ovf2 <= ( not (x1comp(n-1) xor initvalue(n-1))) and  (y2comp(n-1)
xor x1comp(n-1)) ;
   ovf3 <= ( not (x1comp(n-1) xor y1comp(n-1))) and  (z1(n-1) xor
x1comp(n-1)) ;

   y1compt <= y1comp when ovf1='0' else "011111" when initvalue(n-
1)='0' else "100001";
   y2compt <= y2comp when ovf2='0' else "011111" when initvalue(n-
1)='0' else "100001";
   z2 <= z1 when ovf3='0' else "011111" when x1comp(n-1)='0' else
"100001";

g1: SG2COMP port map(y1compt,ytemp1);
g2: SG2COMP port map(y2compt,ytemp2);
g3: SG2COMP port map(z2, z);

output:process(clk)
begin
   if (clk='1' and clk'event) then
     if (load = '1') then
         y1<= initialc;
         y2<= initialc;
     else
       y1<= ytemp1;
       y2<= ytemp2;
     end if;
   end if;
 end process ;

END VNU;
```

A.5.2 VNU3

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

ENTITY Variable_Node3 IS
generic ( n : natural :=6);
```

```
PORT (clk,load:in std_logic; initial,x1,x2,x3 :in std_logic_vector(n-1
downto 0);
     y1,y2,y3, z : out std_logic_vector(n-1 downto 0));
END Variable_Node3;

ARCHITECTURE VNU OF Variable_Node3 IS

component SG2COMP
generic ( n : natural :=6);
    port (x : in std_logic_vector(n-1 downto 0);
          y : out std_logic_vector(n-1 downto 0));
end component;
 signal x1comp,x2comp,x3comp,y1comp,y2comp,y3comp,ytemp1,ytemp2,ytemp3,
y1comp1, y1comp11, y1comp2, y2comp1, y2comp11, y2comp2,
y3comp1,y3comp11,  y3comp2 : std_logic_vector(n-1 downto 0);
 signal initvalue, initialc, z1, z2 : std_logic_vector(n-1 downto 0);
 signal ovf11, ovf12, ovf21, ovf22, ovf31, ovf32, ovfz : std_logic;


BEGIN

gi: SG2COMP port map (initial, initialc);

init: process(clk)
begin
if (load='1' and clk'event and clk='1' ) then
   initvalue<= initialc;
end if;
end process;

u1: SG2COMP port map(x1, x1comp);
u2: SG2COMP port map(x2, x2comp);
u3: SG2COMP port map(x3, x3comp);

   y1comp1<= x2comp+initvalue;
   ovf11 <= ( not (x2comp(n-1) xor initvalue(n-1))) and  (y1comp1(n-1)
xor x2comp(n-1));
   y1comp11 <= y1comp1 when (ovf11='0') else "011111" when initvalue(n-
1)='0' else "100001";
   y1comp2<= y1comp11+x3comp;
   ovf12 <= ( not (x3comp(n-1) xor y1comp11(n-1))) and  (y1comp2(n-1)
xor x3comp(n-1));
   y1comp <= y1comp2 when (ovf12='0') else "011111" when x3comp(n-
1)='0' else "100001";




   y2comp1<= x1comp+initvalue;
   ovf21 <= ( not (x1comp(n-1) xor initvalue(n-1))) and  (y2comp1(n-1)
xor x1comp(n-1));
   y2comp11 <= y2comp1 when (ovf11='0') else "011111" when initvalue(n-
1)='0' else "100001";
   y2comp2<= y2comp11+x3comp;
   ovf22 <= ( not (x3comp(n-1) xor y2comp11(n-1))) and  (y2comp2(n-1)
xor x3comp(n-1));
```

```
   y2comp <= y2comp2 when (ovf22='0') else "011111" when x3comp(n-
1)='0' else "100001";


   y3comp1<= x1comp+initvalue;
   ovf31 <= ( not (x1comp(n-1) xor initvalue(n-1))) and  (y3comp1(n-1)
xor x1comp(n-1));
   y3comp11 <= y3comp1 when (ovf11='0') else "011111" when initvalue(n-
1)='0' else "100001";
   y3comp2<= y3comp11+x2comp;
   ovf32 <= ( not (x2comp(n-1) xor y3comp11(n-1))) and  (y3comp2(n-1)
xor x2comp(n-1));
   y3comp <= y3comp2 when (ovf32='0') else "011111" when x2comp(n-
1)='0' else "100001";

   z1 <= y1comp + x1comp;
   ovfz <= ( not (x1comp(n-1) xor y1comp(n-1))) and  (z1(n-1) xor
x1comp(n-1)) ;
   z2 <= z1 when ovfz='0' else "011111" when x1comp(n-1)='0' else
"100001";


g1: SG2COMP port map(y1comp,ytemp1);
g2: SG2COMP port map(y2comp,ytemp2);
g3: SG2COMP port map(y3comp,ytemp3);
g4: SG2COMP port map(z2, z);

output:process(clk)
begin
   if (clk='1' and clk'event) then
      if (load = '1') then
                  y1<= initialc;
               y2<= initialc;
               y3<= initialc;
      else
               y1<= ytemp1;
               y2<= ytemp2;
               y3<= ytemp3;
        end if;
   end if;
 end process ;
END VNU;
```

A.7 The Number Format Conversion Files

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity SG2COMP is
generic ( n : natural :=6);
PORT ( x: in  std_logic_vector(n-1 downto 0);
      y : out std_logic_vector(n-1 downto 0));
```

```
END SG2COMP ;

Architecture behavioral of SG2COMP is
 begin
  process(x)
   variable sign: std_logic;
   variable mag: std_logic_vector(n-2 downto 0);
  begin
     sign := x(n-1);
      if sign = '1' then
         mag := not x(n-2 downto 0) + 1;
      else
         mag := x(n-2 downto 0);
      end if;
  y <= sign & mag ;
  end process;

 end behavioral;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity tctosm is
generic ( n : natural :=5);
port (input: in std_logic_vector(n-1 downto 0);
      output: out std_logic_vector(n-1 downto 0)
       );
end tctosm;

architecture Behavioral of tctosm is

begin
  process(input)
   variable sign: std_logic;
   variable mag: std_logic_vector(n-2 downto 0);
  begin
     sign := input(n-1);
      if sign = '1' then
         mag := not input(n-2 downto 0) + 1;
      else
         mag := input(n-2 downto 0);
      end if;
  output <= sign & mag ;
  end process;


end Behavioral;
```

Appendix B
MATLAB Code for Generating H matrix of the proposed code

The following list includes the MATLAB code used for generating **H** matrix for any sample of the LDPC code proposed in Chapter 4. It contains all the possible connections in all of the five levels: the intra-cell connections, the horizontal connections (levels 1&2) and the vertical connections (levels 1&2). By removing different varieties of connections, we get new codes.

```
% A Program that will generate the H matrix for the proposed LDPC code.

%***** Defining the dimensions of the structure****

% The number of rows of ministructures
P=8 ;
% The number of columns of ministructures
Q=8;

% The size of the code
n=P*Q*16 ;
k=P*Q*8 ;

% Initializing the H matrix
H=zeros(n-k,n);

%****** Inner Connections ********
for x=1:P
    for y=1:Q
        H(1+((x-1)*Q+(y-1))*8,1+((x-1)*Q+(y-1))*16)=1;
        H(1+((x-1)*Q+(y-1))*8,2+((x-1)*Q+(y-1))*16)=1;
        H(1+((x-1)*Q+(y-1))*8,5+((x-1)*Q+(y-1))*16)=1;
        H(1+((x-1)*Q+(y-1))*8,15+((x-1)*Q+(y-1))*16)=1;
        H(2+((x-1)*Q+(y-1))*8,2+((x-1)*Q+(y-1))*16)=1;
        H(2+((x-1)*Q+(y-1))*8,3+((x-1)*Q+(y-1))*16)=1;
        H(2+((x-1)*Q+(y-1))*8,6+((x-1)*Q+(y-1))*16)=1;
        H(2+((x-1)*Q+(y-1))*8,16+((x-1)*Q+(y-1))*16)=1;
        H(3+((x-1)*Q+(y-1))*8,3+((x-1)*Q+(y-1))*16)=1;
        H(3+((x-1)*Q+(y-1))*8,4+((x-1)*Q+(y-1))*16)=1;
        H(3+((x-1)*Q+(y-1))*8,7+((x-1)*Q+(y-1))*16)=1;
        H(3+((x-1)*Q+(y-1))*8,13+((x-1)*Q+(y-1))*16)=1;
        H(4+((x-1)*Q+(y-1))*8,1+((x-1)*Q+(y-1))*16)=1;
        H(4+((x-1)*Q+(y-1))*8,4+((x-1)*Q+(y-1))*16)=1;
        H(4+((x-1)*Q+(y-1))*8,8+((x-1)*Q+(y-1))*16)=1;
        H(4+((x-1)*Q+(y-1))*8,14+((x-1)*Q+(y-1))*16)=1;
        H(5+((x-1)*Q+(y-1))*8,9+((x-1)*Q+(y-1))*16)=1;
        H(5+((x-1)*Q+(y-1))*8,10+((x-1)*Q+(y-1))*16)=1;
        H(5+((x-1)*Q+(y-1))*8,13+((x-1)*Q+(y-1))*16)=1;
```

```
        H(5+((x-1)*Q+(y-1))*8,5+((x-1)*Q+(y-1))*16)=1;
        H(6+((x-1)*Q+(y-1))*8,6+((x-1)*Q+(y-1))*16)=1;
        H(6+((x-1)*Q+(y-1))*8,10+((x-1)*Q+(y-1))*16)=1;
        H(6+((x-1)*Q+(y-1))*8,11+((x-1)*Q+(y-1))*16)=1;
        H(6+((x-1)*Q+(y-1))*8,14+((x-1)*Q+(y-1))*16)=1;
        H(7+((x-1)*Q+(y-1))*8,7+((x-1)*Q+(y-1))*16)=1;
        H(7+((x-1)*Q+(y-1))*8,11+((x-1)*Q+(y-1))*16)=1;
        H(7+((x-1)*Q+(y-1))*8,12+((x-1)*Q+(y-1))*16)=1;
        H(7+((x-1)*Q+(y-1))*8,15+((x-1)*Q+(y-1))*16)=1;
        H(8+((x-1)*Q+(y-1))*8,8+((x-1)*Q+(y-1))*16)=1;
        H(8+((x-1)*Q+(y-1))*8,12+((x-1)*Q+(y-1))*16)=1;
        H(8+((x-1)*Q+(y-1))*8,9+((x-1)*Q+(y-1))*16)=1;
        H(8+((x-1)*Q+(y-1))*8,16+((x-1)*Q+(y-1))*16)=1;
    end
end

%****** Horizontal Connections ********
for x=1:P
    for y=1:Q-1
        H(9+((x-1)*Q+(y-1))*8,5+((x-1)*Q+(y-1))*16)=1;
        H(10+((x-1)*Q+(y-1))*8,6+((x-1)*Q+(y-1))*16)=1;
        H(11+((x-1)*Q+(y-1))*8,7+((x-1)*Q+(y-1))*16)=1;
        H(12+((x-1)*Q+(y-1))*8,8+((x-1)*Q+(y-1))*16)=1;
        H(13+((x-1)*Q+(y-1))*8,14+((x-1)*Q+(y-1))*16)=1;
        H(14+((x-1)*Q+(y-1))*8,13+((x-1)*Q+(y-1))*16)=1;
        H(15+((x-1)*Q+(y-1))*8,16+((x-1)*Q+(y-1))*16)=1;
        H(16+((x-1)*Q+(y-1))*8,15+((x-1)*Q+(y-1))*16)=1;
    end
end

%****** Vertical Connections ********

for x=2: P
  for y=1: Q
   H(1+((x-1)*Q+(y-1))*8,1+((x-2)*Q+(y-1))*16)=1;
   H(2+((x-1)*Q+(y-1))*8,9+((x-2)*Q+(y-1))*16)=1;
   H(3+((x-1)*Q+(y-1))*8,4+((x-2)*Q+(y-1))*16)=1;
   H(4+((x-1)*Q+(y-1))*8,12+((x-2)*Q+(y-1))*16)=1;
   H(5+((x-1)*Q+(y-1))*8,10+((x-2)*Q+(y-1))*16)=1;
   H(6+((x-1)*Q+(y-1))*8,2+((x-2)*Q+(y-1))*16)=1;
   H(7+((x-1)*Q+(y-1))*8,11+((x-2)*Q+(y-1))*16)=1;
   H(8+((x-1)*Q+(y-1))*8,3+((x-2)*Q+(y-1))*16)=1;
    end
end

%****** Horizontal Connections  2  ********

if Q >= 3
    for x = 1:P
        for y = 1:Q-2
           H( (1) + ((x-1)*Q+(y-1)) * 8 , (37) + ((x-1)*Q+(y-1)) * 16
)=1;
            H( (2) + ((x-1)*Q+(y-1)) * 8 , (45) + ((x-1)*Q+(y-1)) * 16
)=1;
            H( (3) + ((x-1)*Q+(y-1)) * 8 , (46) + ((x-1)*Q+(y-1)) * 16
)=1;
```

```
            H( (4) + ((x-1)*Q+(y-1)) * 8 , (48) + ((x-1)*Q+(y-1)) * 16
)=1;
             H( (5) + ((x-1)*Q+(y-1)) * 8 , (39) + ((x-1)*Q+(y-1)) * 16
)=1;
             H( (6) + ((x-1)*Q+(y-1)) * 8 , (40) + ((x-1)*Q+(y-1)) * 16
)=1;
             H( (7) + ((x-1)*Q+(y-1)) * 8 , (38) + ((x-1)*Q+(y-1)) * 16
)=1;
             H( (8) + ((x-1)*Q+(y-1)) * 8 , (47) + ((x-1)*Q+(y-1)) * 16
)=1;
         end
    end
end

%****** Vertical Connections   2  ********

if P >= 3
     for x = 1:P-2
         for y = 1:Q
             H( (1) + ((x-1)*Q+(y-1)) * 8, (4) + ( (x+1) *Q+(y-1)) * 16
)=1;
            H( (2) + ((x-1)*Q+(y-1)) * 8, (12) + ( (x+1) *Q+(y-1)) * 16
)=1;
            H( (3) + ((x-1)*Q+(y-1)) * 8, (12) + ( (x+1) *Q+(y-1)) * 16
)=1;
             H( (4) + ((x-1)*Q+(y-1)) * 8, (1) + ( (x+1) *Q+(y-1)) * 16
)=1;
             H( (5) + ((x-1)*Q+(y-1)) * 8, (12) + ( (x+1) *Q+(y-1)) * 16
)=1;
             H( (6) + ((x-1)*Q+(y-1)) * 8, (2) + ( (x+1) *Q+(y-1)) * 16
)=1;
             H( (7) + ((x-1)*Q+(y-1)) * 8, (9) + ( (x+1) *Q+(y-1)) * 16
)=1;
           H( (8) + ((x-1)*Q+(y-1)) * 8, (12) + ( (x+1) *Q+(y-1)) * 16
)=1;
         end
     end
end
```

References

[1] Gallager, R.G. *Low-density parity-check codes.* Cambridge, MA: MIT Press, 1963.

[2] Mackay, D.J.C, and R.*M*. Neal. "Near Shannon limit performance of low density parity check codes." *Electron. Lett., vol. 33, no. 6*, Mar 1997: 457–458.

[3] Chung, S.Y., D.J. Forney, T.J Richardson, and R.L. Urbanke. "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit." *IEEE Communications Letters* 5, no. 2 (Feb. 2001): 58–60.

[4] Boutillon, E., J. Castura, and F.R. Kschischang. "Decoder first code design." *Proc. Int. Symp. on Turbo Codes and Related Topics.* Brest, France, 2000. 459–462.

[5] Echard, R., and S.C. Chang. "The π-rotation low-density parity check codes." *Proc. GLOBECOM 2001, vol. 2.* San Antonio, TX, 2001. 980–984.

[6] Fossorier, *M*., *M*. Mihaljevic, and H. Imai. "Reduced Complexity Iterative Decoding of Low-Density Parity Check Codes Based on Belief Propagation." (IEEE TRANSACTIONS ON COMMUNICATIONS) 47, no. 5 (May 1999).

[7] Zhang, T, and K K Parhi. "A 56 Mbps (3,6)-regular FPGA LDPC Decoder." *Workshop on Signal Processing Systems.* SIPS, 2002.

[8] Chen, Y., and D. Hocevar. "A FPGA and ASIC Implementation of Rate 1/2, 8088-b Irregular Low Density Parity Check Decoder." *IEEE Global Telecommunications Conference.* GLOBECOM ’03, 2003. 1-5.

[9] Blanksby, and C. Howland. "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-

check code decoder,." *IEEE Journal of Solid-State Circuits* 37 (Mar. 2002): 404–412.

[10] Mansour, *M.M*., and N.R. Shanbhag. "Low-Power VLSI Decoder Architectures for LDPC Codes." *Int. Symp. Low Power Electronic Design.* Aug. 2002. 12-14.

[11] Mansour, *M.M*., and N.R. Shanbhag. "Turbo decoder architectures for low-density parity-check codes." *IEEE Global Telecommunications Conference.* GLOBECOM '02, Nov. 2002. 17 - 21.

[12] Bhatt, T., K. Narayanan, and N. Kehtarnavaz. "Fixed-point DSP Implementation of Low-Density Parity Check Codes." *9th DSP (DSP 2000) Workshop.* Oct. 2000.

[13] Eleftheriou, E., and S. Olcer. "Low-Density Parity-Check Codes for Digital Subscriber Lines." *IEEE International Conference on Communications.* ICC 2002, 2002.

[14] Sorokine, V., F. Kschischang, and S. Pasupathy. "Gallager Codes for CDMA Applications I: Generalizations, Constructions and Performance Bounds." (IEEE Transactions on Communications) 48 (Oct. 2000).

[15] Futaki, Hisashi, and Tomoaki Ohtsuki. "Low-Density Parity-Check (LDPC) Coded OFDM Systems with *M*-PSK." VTC 2002, 2002.

[16] Sorokine, Vladislav, Frank Kschischang, and S. Pasupathy. "Gallager Codes for CDMA Applications: Generalizations, Constuctions and Perfromance." Killarney, Ireland: ITW 1998, 1998.

[17] Richardson, T., and R. Urbanke. "The capacity of low-density parity-check codes

under message passing decoding." *IEEE Trans. on Inform. Theory, 47*, Feb 2001: 599–618.

[18] Futaki, H., and T. Ohtsukif. "Low-density parity-check (LDPC) coded MIMO systems with iterative turbo decoding." *IEEE 58th Vehicular Technology Conference.* Tokyo, Japan: IEEE, 2003.

[19] Lu, Ben, Xiaodong Wang, and K. Narayanan. "LDPC-Based Space-Time Coded OFDM Systems Over Correlated Fading Channels:Performance Analysis and Receiver Design." (IEEE Transactions on Communications) 50, no. 1 (Jan. 2002).

[20] Sridharan, A., D Costello, D. Sridhara, T. Fuja, and R Tanner. "A Construction for Low Density Parity Check Convolutional Codes Based on Quasi-Cyclic Block Codes." Lausanne, Switzerland: ISIT 2002, 2002.

[21] Levine, B., R.R. Taylor, and H. Schmit. "Implementation of near Shannon limit error-correcting codes using reconfigurable hardware." *IEEE Symposium on Field-Programmable Custom Computing Machines.* 2000. 217–226.

[22] Sorokine, V., F. Kschischang, and S. Pasupathy. "Gallager Codes for CDMA Applications II: Implementations, Complexity and System Capacity." (IEEE Transactions on Communications) 48 (Nov. 2000).

[23] Richardson, T., A. Shokrollahi, and R. Urbanke. "Design of capacity approaching irregular low density parity check codes." *IEEE Trans. on Inform. Theory, 47*, Feb 2001: 618–637.

[24] McGowan, J. A., and R.C. Williamson. "Loop removal from LDPC codes."

Proceedings of IEEE Information Theory Workshop. 2003. 230- 233.

[25] Kou, Y., S. Lin, and *M*. Fossorier. "Low-density parity-check codes based on finite geometries: a rediscovery and new results." *IEEE Trans. Inform. Theory, 47*, 2001: 2711–2736.

[26] Ping, W.K. Leung, and W. Phamdo. "Low density parity check codes with semi-random parity check matrix." *Electron. Lett., vol. 35, no. 1*, Jan. 1999: pp. 38–39.

[27] Mackay, D. " Good error correcting codes bases on very sparse matrices." *IEEE Trans. on Inform. Theory, 45*, Mar 1999: 399–431.

[28] Tang, H., J. Xu, S. Lin, and K. Abdel-Ghaffar. "Codes on finite geometries." (IEEE Trans. Inf. Theory) 2002.

[29] Xiao, H., and A. Banihashemi. "Message-Passing Schedule for Decoding LDPC Codes." (IEEE Trans. Comm) 2003.

[30] Gallager, R. "Low-Density Parity-Check Codes." *IRE Trans. Information Theory*, Jan 1962: 21-28.

[31] Zhang, Wei, Guangxi Zhu, Li Peng, and Qiongxia Shen. "Design of Low-Complexity Well-Structured LDPC Codes Based on Iterative-Filled Approach." *Congress on Image and Signal Processing, 2008. CISP apos;08.* . 27-30 May 2008. 209 - 213.

[32] Mohiyuddin, Marghoob, Amit Prakash, Adnan Aziz, and Wayne Wolf. "Synthesizing interconnect-efficient low density parity check codes." *Proceedings of the 41st annual conference on Design automation* . San Diego, CA, USA : ACM,

2004. 488 - 491.

[33] El-Maleh, Aiman, Basil Arkasosy, and Adnan Al-Andalusi. "Interconnect-Efficient LDPC Code Design." *18th IEEE Int.Conf. on Microlelectronics.* Dec. 2006. 127-130.

[34] Alghonaim, Esa, Aiman El-Maleh, and *M.*Adnan Landolsi. "New Techniques for Improving Performance of LDPC Codes in the Presence of Trapping Sets." 2007.

[35] Richardson, Tom, and Rudiger Urbanke. "The Renaissance of Gallager's Low-Density Parity-Check Codes." *IEEE Communications Magazine*, Aug. 2003: 126-131.

[36] Yeo, Engling, Borivoje Nikolic, and Venkat Anantharam. "Iterative Decoder Architectures." *IEEE Communications Magazine*, Aug. 2003: 132-140.

[37] Chung, S.Y., T. Richardson, and R Urbanke. "Analysis of sum-product decoding of low-density parity-check codes using a gaussian approximation." *IEEE Trans. on Inform. Theory,47*, Feb 2001: 657–670.

[38] Fan, John. *Constrained Coding and Soft Iterative Decoding.* Springer, 2001.

[39] Kschischang, F.R., B.J. Frey, and H.A. Loeliger. "Factor graphs and the sum-prodcut algorithm." *IEEE Trans. on Inform. Theory, 47*, Feb 2001: 498–519.

[40] Howland, C., and A. Blanksby. "A 220mW 1Gb/s 1024-Bit Rate-1/2 Low Density Parity Check Code Decoder." *The 2001 IEEE International Symposium on Circuits and Systems, Volume: 4.* ISCAS 2001, May 2001. 6-9.

[41] Zhang, T., and K.K. Parhi. "Joint (3,k)-regular LDPC code and decoder/encoder

design." *IEEE Trans. Sig. Proc., vol. 52, no. 4*, Apr. 2004: 1065–1079.

[42] Yeo, E., B Nikolic, and V. Anantharam. "High Throughput Low-Density Parity-Check Decoder Architectures." *IEEE Globecom 2001*. San Antonio, Nov. 2001. 25-29.

[43] Ryan, W.E. "An Introduction to LDPC Codes." In *CRC Handbook for Coding and Signal Processing for Recording Systems*, by B. Vasic. CRC Press, 2004.

[44] Richardson, T.J., A. Shokrollahi, and R. Urbanke. "Design of provably good lowdensity parity-check codes." *Proc. Int. Symp. Information Theory*. Sorrento, Italy, 2000. 199.

[45] Luby, *M*.G., *M*.A. Shokrolloahi, *M*. Mizenmacher, and D.A. Spielman. "Improved low-density parity-check codes using irregular graphs." *IEEE Trans. Inform. Theory, vol. 47, no. 2*, Feb. 2001: 585–598.

[46] Howland, C., and A. Blanksby. "Parallel Decoding Architectures for Low Density Parity Check Codes." *IEEE Conference on Custom Integrated Circuits*. May 2001b.

[47] He, Zhiyong, Sebastien Roy, and Paul Fortier. "FPGA Implementation of LDPC Decoders Based on Joint Row-column Decoding Algorithm." *IEEE International Symposium on Circuits and Systems, 2007. ISCAS 2007.* . New Orleans, LA, 27-30 May 2007. 1653-1656.

[48] Masera, G, F Quaglio, and F Vacca. "Implementation of a Flexible LDPC Decoder." *IEEE Transactions on Circuits and Systems II: Express Briefs* 54, no. 6 (June 2007): 542 - 546.

[49] McDonagh, J, *M* Sala, V Katewa, and E Popovici. "Efficient construction and implementation of short LDPC codes for wireless sensor networks." *18th European Conference on Circuit Theory and Design, 2007. ECCTD 2007.* 2007. 703 - 706.

[50] Dore, J.B; Penard, P; Hamon, M.H. "Architecture and Design Methodology for Structured LDPC Decoder." *IEEE 66th Vehicular Technology Conference, 2007. VTC-2007* . 2007. 1142 - 1146.

[51] Seo, Sangwon, Trevor Mudge, Yuming Zhu, and Chaitali Chakrabarti. "Design and Analysis of LDPC Decoders for Software Defined Radio." *2007 IEEE Workshop on Signal Processing Systems.* Shanghai, China, 17-19 Oct. 2007. 210-215.

[52] Patil, S.R., and S.S. Pathak. "Construction of irregular LDPC codes based on Balanced Incomplete Block Designs." *International Conference Industrial and Information Systems, 2007. ICIIS 2007.* 9-11 Aug. 2007. 231-234.

[53] Guilloud, F, E Boutillon, J. Tousch, and J. Danger. "Generic Description and Synthesis of LDPC Decoders." *IEEE Transactions on Communications* 55, no. 11 (Nov. 2007): 2084 - 2091.

[54] Masera, G., F. Quaglio, and F. Vacca. "Finite precision implementation of LDPC decoders." *IEE Proceedings in Communications* 152, no. 6 (Dec. 2005): 1098- 1102.

[55] Ku, Mong-Kai, Huan-Sheng Li, and Yi-Hsing Chien. "Code design and decoder implementation of low density parity check code." *Emerging Information Technology Conference, 2005.* 15-16 Aug. 2005.

[56] Beuschel, C., and H. Pfleiderer. "FPGA implementation of a flexible decoder for

long LDPC codes." *International Conference on Field Programmable Logic and Applications, 2008. FPL 2008.* Heidelberg, 8-10 Sept. 2008. 185-190.

[57] Dore, J., *M* Hamon, and P. Penard. "On Flexible Design and Implementation of Structured LDPC Codes." *IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications, 2007. PIMRC 2007.* 3-7 Sept. 2007. 1-5.

[58] Zhu, Y., and C. Chakrabarti. "Architecture-Aware LDPC Code Design for Software Defined Radio." *IEEE Workshop on Signal Processing Systems Design and Implementation, 2006. SIPS '06.* 2-4 Oct. 2006. 405-410.

[59] Brack, T., et al. "Low Complexity LDPC Code Decoders for Next Generation Standards." *Design, Automation & Test in Europe Conference & Exhibition, 2007.* 16-20 April 2007. 1 - 6.

[60] ALGHONAIM, ESA. *Techniques for Improving LDPC Codes Performance [Dissertation]*. KFUPM, March 2008.